1977-03

A shared environment for microcomputer system development

Brown, Kenneth J.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

A SHARED ENVIRONMENT FOR
MICROCOMPUTER SYSTEM DEVELOPMENT

by

Kenneth J. Brown
and
David R. Bullock

March 1977

Thesis Advisor: G. M. Raetz

Approved for public release; distribution unlimited.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) A Shared Environment for Microcomputer System Development	5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; March 1977
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kenneth J. Brown and David R. Bullock	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940	12. REPORT DATE March 1977
	13. NUMBER OF PAGES 220
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

microcomputer
operating system
resource sharing
microprocessor
timesharing

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A timeshared microcomputer monitor for Intel 8080 microprocessor systems development has been described. Running on the Sycor 440 Clustered Terminal Processing System, the monitor provides a cirtual environment composed of a console device, eight floppy disk drives, and an 8080 microprocessor for up to four concurrent users. Virtual floppy disk files on a five megabyte movable-head disk provide the system's

(cont.)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20. (cont.)

primary auxiliary storage medium. Three different levels of access protection are available for these virtual floppy disk images. A command language processor has been included to support on-line modification of the virtual environment. System recovery in the event of a hardware or software failure is also supported by the monitor.

Approved for public release; distribution unlimited.

A Shared Environment for Microcomputer System Development

by

Kenneth J. Brown
Captain, United States Marine Corps
B.A., California State College Fullerton, 1968

and

David R. Bullock
Lieutenant, United States Navy
B.A., State University of New York College at Oneonta, 1968

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March, 1977

ABSTRACT

A timeshared microcomputer monitor for Intel 8080 microprocessor systems development has been described. Running on the Sycor 440 Clustered Terminal Processing System, the monitor provides a virtual environment composed of a console device, eight floppy disk drives, and an 8080 microprocessor for up to four concurrent users. Virtual floppy disk files on a five megabyte movable-head disk provide the system's primary auxiliary storage medium. Three different levels of access protection are available for these virtual floppy disk images. A command language processor has been included to support on-line modification of the virtual environment. System recovery in the event of a hardware or software failure is also supported by the monitor.

TABLE OF CONTENTS

TABLE OF ABBREVIATIONS

ASCII	- American Standard Code for Information Interchange
CP/M	- Control Program for Microcomputers
CPU	- central processing unit
CRT	- cathode ray tube
CTSS	- Compatible TimeSharing System
DCB	- disk control block
DMA	- direct memory access
DMT	- disk map table
I/O	- input/output
IPL	- initial program load
K	- tenth power of 2, i.e. 1024
LSI	- large scale integration
MTS	- Microcomputer Timeshared System
NPS	- Naval Postgraduate School, Monterey, Calif.
PL/M	- Programming Language for Microcomputers
PLMR	- PL/M with relocation
RAM	- random access memory
ROM	- read only memory
SPOOL	- simultaneous peripheral operations on line
SSB	- system state block
TCT	- task control table
TTY	- teletypewriter
VMM	- virtual machine monitor

I. INTRODUCTION

A. BACKGROUND

In the summer of 1976 the Computer Science Department at the Naval Postgraduate School (NPS) acquired a Sycor Model 440 Clustered Terminal Processing System for use in the NPS microcomputer laboratory. The Sycor 440 utilizes an Intel 8080 LSI chip as the CPU of a special purpose microcomputer system designed primarily for data entry applications. In addition to the 8080 CPU, the 440 hardware configuration includes a five megabyte movable-head disk, 64K of random access memory (RAM), a cassette tape drive, and four display terminals consisting of a keyboard and cathode ray tube (CRT) display device. Two peripheral devices are also provided: a serial printer and an RS-232 compatible asyncronous communication interface.

As part of the Model 440 System, Sycor, Inc. supplied a comprehensive package of system software. Most of this software was designed to support applications in the data entry field which comprised the Sycor 440's primary marketing target. In addition to the basic data entry package Sycor also supplied several programs which exercise the 440's capabilities more fully. These included a resident COBOL compiler; a cross-compiler for PLMR, a relocatable version of the system development language PL/M;

and a linking loader which combines relocatable segments of COBOL or PLMR object code into an executable object module.

Prior to the arrival of the Sycor 440, the facilities available in the NPS microcomputer laboratory consisted primarily of two INTELLEC 8/MOD 80 microcomputer systems designed for use in the development of software for the Intel 8080 chip processors. Both of these systems support the CP/M (Control Program for Microcomputers) operating system [2]. CP/M provides many software development tools including a context editor, dynamic debugger, and assembler for the generation of 8080 object code [3-6]. However, CP/M provides only a single display terminal interface, and uses floppy disks as its auxiliary storage medium.

While the Sycor 440 made a significant and welcome addition to the hardware complement of the NPS microcomputer laboratory, a great deal of effort was required to integrate the new system into the current laboratory configuration and to ensure that maximum benefit could be derived from the new acquisition. The following paragraphs summarize the goals and objectives of this effort.

B. GOALS AND OBJECTIVES

The primary goal of the Sycor 440 integration effort was to complete an investigation of the design and architecture of the available hardware configuration, and the capabilities of the existing software with an eye towards the most feasible use of the Sycor 440 system in support of

the tutorial and research activities of the NPS microcomputer laboratory. It was felt that the Sycor 440 hardware had the capability to support numerous applications more generalized than the data entry function it was designed and programmed to accomplish. In particular, the cluster of four terminals suggested the development of a timeshared environment to support up to four simultaneous users on a single Sycor 440. Such a timeshared system would provide an effective increase from two to six 8080 CPU's available for use in the microcomputer laboratory. This increase would go a long way towards meeting the increased demand for microcomputer research and development facilitites at NPS.

A secondary goal of the 440 integration effort was to ensure that a well-defined, compatible interface was established between the current laboratory configuration and the newly acquired Sycor system. The objective here was twofold: first, to achieve hardware compatibility between the Sycor 440 and other computer systems available in the lab, and secondly to achieve software compatibility between the Sycor 440 and other systems which utilize the 8080 CPU chip.

Achieving hardware compatibility depended on tailoring the 440's asyncronous communication interface to the requirements of existing systems, particularly the PDP-11/50 which served as the central interface point for inter-processor communication. It was anticipated that this hardware interface could be easily established without

modification of the Sycor system.

Achieving software compatibility was viewed as a much more difficult objective. This would involve developing an environment which could support software that had already been written for other systems utilizing the 8080 CPU chip. For programs developed at NPS this meant that the Sycor 440 environment must support a suitably modified version of CP/M.

C. PROBLEM DEFINITION

The thesis project which eventually evolved from the Sycor 440 integration effort was development of the Microcomputer Timeshared System (MTS), a software monitor for the Sycor 440 designed to support microcomputer system development. The possibility of providing a virtual machine environment was investigated, including the incorporation of virtual device interfaces where practical. The Sycor 440 file system was to be utilized where possible in order to reduce development time, avoid duplication of system facilities, and allow the Sycor operating system to be used. The man-machine interface at the terminals was to be simple, flexible, convenient and incorporate the best features of existing interactive systems at NPS. This system was to provide a suitable interface for user programs and/or operating systems to enable utilization of the Sycor 440 storage and peripheral devices by these programs.

It was realized from the project's inception that a

microcomputer timesharing system is ideally implemented through multiprocessing rather than multiprogramming. The use of LSI technology has made the CPU cost only a minor factor in the total cost of a microcomputer system. A major objective in the development of MTS was finding an efficient, practical method of sharing hardware resources other than the CPU. The unit costs of memory, fixed disks, printers, terminals, and other peripherals are major factors in total system cost - the sharing of these resources was the primary consideration.

To achieve the project objectives, a survey of timesharing concepts and implementations was conducted. Particular attention was devoted to the current design and implementation techniques of virtual machine timesharing systems to determine those features which could be included in MTS. This research included study of timesharing system techniques associated with processor management, memory management, and device management. Those factors which influenced the design of MTS are discussed in the next chapter.

II. TIMESHARING CONCEPTS

The concept of timesharing computer system resources was first demonstrated in the early 1960's. Since that time, growth in the application of timesharing concepts to many classes of system operational requirements and designs has occurred rapidly. The timesharing field has matured now to the point that one can recognize many common design goals and implementation concepts being used in a wide variety of systems. General characteristics of timesharing systems, methods of implementation, and resource management techniques were researched in order to identify concepts which could be applied to a microcomputer timeshared system.

A. CHARACTERISTICS OF TIMESHARING SYSTEMS

One of the original motivations for development of early timesharing systems was to obtain more efficient use of a computer system's expensive operating time and resources than was being realized through batch environments. Interactive or conversational timesharing imposes the additional burden of keeping the "user" busy, as well as the hardware. System response time must be directly proportional to the user's expectations. Those tasks which the user perceives as simple must result in rapid response from the system. For example, character echoing or input

line editing might be expected to require no appreciable delay. On the other hand, the user would expect a large program assembly or compilation to take more time and therefore some delay would be acceptable.

Another important consideration is that of system protection. The independence of concurrently executing processes must be maintained. Protection of the system from user processes, as well as user processes from each other, must be considered. Protection of system resources such as the file system and I/O devices must be maintained. System protection may be implemented at many levels, both in hardware and software. Within the software system, protection information for specific resources is normally maintained in various tables associated with these resources [16]. Hardware mechanisms can be provided to support system protection. For example, the use of bound registers to trap invalid memory references could be implemented [10]. The operating system can be insulated from the user processes by providing separate system states (e.g. system and user) or other lockout mechanisms to identify when the system is executing in the system rather than the user state.

Related to system protection are the important concepts of system reliability and recoverability. The user expects the system to operate reliably and if a failure should occur, to recover as smoothly and rapidly as possible. That is, recovery should preserve the user data and programs which were not related to the failure. The goal in designing recovery procedures is to minimize the impact of

the recovery mechanism on the system when it is operating normally, yet ensure smooth and rapid recovery should a failure occur.

B. VIRTUAL MACHINES

In recent years, the multiplexing of computer system resources has been accomplished in some implementations by utilizing the Virtual Machine Monitor (VMM) concept. A VMM is a special form of operating system that multiplexes only the physical resources among the users - no other functional enhancements are provided [10]. The concept is to produce the effect of multiple bare machines which are identical to the bare machine on which the VMM runs. Thus, each user can select the operating system of his choice to run on his "private" computer.

1. Virtual Devices

Under the VMM concept, all I/O is simulated. When a user program attempts to execute an I/O instruction, control is tranferred to the VMM via an interrupt. There are normally three different situations which may arise:

(1) If the I/O device physically exists and has been assigned to that user program, the instruction may be executed immediately without modification.

(2) If a similar I/O device exists, the I/O commands are appropriately modified and then executed. For example, many small disks may be simulated by using separate areas of a single large disk.

(3) Certain devices, such as printers and card readers, may be extensively simulated using techniques such as spooling (see Device Management) [10].

2. Hardware Requirements

The implementation of a practical virtual machine monitor requires that the host computer have certain hardware characteristics:

(1) The instruction set should contain both privileged and non-privileged instructions.

(2) The host computer must have some way of distinguishing between the two types of instructions. That is, the VMM must be made aware of any attempt by a user's program to execute a privileged instruction, or change its mode of operation. This is normally satisfied by establishing two separate states of operation, system and user.

(3) The VMM must be protected from the user programs, i.e. the memory assigned to the VMM must be protected in both read and write mode.

(4) The user programs must be protected or isolated from each other. This means that memory or input/output devices assigned to one user program must be inaccessible from any other user program, unless proper safeguards are provided. Isolation and protection of memory areas can be accomplished by some form of memory protection or an address translation scheme [11].

A machine which satisfies these requirements can ensure the degree of control by the VMM necessary to avoid excessive overhead in the translation or simulation process.

C. PROCESSOR MANAGEMENT

The task of processor management in the multiprogramming environment of a timeshared system involves the scheduling and managing of multiple processes in different stages of completion. A process may exist in one of three states: (1) running, (2) ready to execute, or (3) blocked pending the completion of I/O or some other process.

A process is in the running state if it is executing instructions. The ready to execute (ready) state describes a process that is ready for execution but not currently running due to the unavailability of the CPU. A blocked process is one which cannot run until some signal arrives to unblock it. These unblocking signals are referred to as "wake-up" signals and change the status of a process from blocked to the ready state. Such signals can come from many sources: system processes, user processes, hardware interrupts, such as terminal communications equipment, a timer, or completion of a disk I/O operation [16].

Since there is no guarantee that a process will block itself, timesharing systems must provide a mechanism for regaining control of the CPU from the currently running process in order to provide all users with adequate response times. The length of time which a process is allowed to run

before it is blocked is called a quantum or a timeslice. The quantum size is one of the most important parameters of a scheduling algorithm. It may be fixed or variable, depending on other parameters such as process size, process priority, or length of time the process last ran. Setting quantum lengths is a function of the system characteristics and workload, and can be determined only by experiment with the actual system or by simulation [16].

D. MEMORY MANAGEMENT

Memory management involves the memory allocation and swapping functions. This includes keeping track of the virtual memory space of each process in the system, whether it is in main memory, auxiliary storage, or both. The complexity of this task may range from maintaining relatively few tables in support of a single contiguous allocation scheme, to the maintaining of many memory allocation data structures to support a virtual memory implementation using demand-paging or segmented memory management schemes [10,16].

Swapping can be defined as moving processes between main memory and auxiliary storage in order to multiplex main memory. Swapping time is the time it takes to complete a swapping task. The multiprogramming of processes in a timeshared system results in the swapping time having a major impact on the system's response time. There are two important parameters of an auxiliary storage device that

affect swapping times:

(1) The average length of time required to access the required block of information. This is called the average access time.

(2) The time required to transfer the block to and from main memory. This is inversely proportional to the transfer rate [16].

Early timesharing systems such as the Compatible TimeSharing System (CTSS) used very simple allocation and swapping strategies [10]. No attempt was made to overlap the execution of one process with the swapping of another. Only one complete process resided in memory at once and all processes were loaded relative to a constant fixed location. That is, no dynamic relocation was used because no dynamic relocation hardware was available on these early systems.

E. DEVICE MANAGEMENT

Device management involves keeping track of the device resources, allocating the device resources to a process, initiating the I/O operation and reclaiming the resource (in most cases the I/O terminates automatically). Devices fall into two general categories:

(1) Dedicated - those devices which are most efficiently assigned to one user for a given time period, even though the user may not be able to utilize the device continuously. In this category are tape drives, printers, and card equipment.

(2) Sharable - those devices which, while allowing access to only one process at a time, can rapidly complete their service for individual processes and be quickly switched to service requests of other processes. In this category are such online auxiliary storage units as drums, disks, and data cells [16].

The operation of some dedicated devices may be simulated to provide more flexibility and improved responsiveness to user requests. For example, the operation of printing a file on the printer could be transformed into a "write" onto a disk (a virtual printer) where at some later time a second routine would copy the information onto the actual printer. This process is called spooling, and allows (1) dedicated devices to be shared, hence, more flexibility in scheduling these devices; (2) more flexibility in job scheduling; and (3) improved synchronization of the speed of the device and the arrival rate of requests for that device [10].

III. SYCOR 440 HARDWARE DESCRIPTION

The Sycor 440 Clustered Terminal Processing System at NPS is composed of a control unit containing a cassette tape drive, four display terminals, a Centronix serial printer, and a Sycor Model 340 Communications Terminal.

The control unit is the heart of the 440 system. Contained within a waist-high cabinet are random and control logic including two 8080 chips, 64K of random access memory (RAM), interfaces for all peripheral devices, a five megabyte disk, as well as the cassette tape drive.

One of the two 8080 chips located in the 440 control unit serves as the system CPU. The 8080 instruction set consists of 78 data transfer, arithmetic, logical, branch, stack, I/O, and machine control instructions [8]. The Sycor 440 provides a comprehensive set of prioritized interrupts including a timer, peripheral device, and auxiliary storage device interrupts. Control information and data are passed between the 8080 CPU and peripheral devices through the I/O ports (referred to as latches in Sycor literature) provided on the 8080 chip.

The second 8080 chip found in the control unit acts as a controller for the mini-disk. The mini-disk is a single platter, movable head disk blocked into 512 byte sectors. There are 800 tracks on the disk with 13 sectors per track. Data transfer between RAM and the mini-disk is via direct

memory access (DMA). The mini-disk controller communicates with the host 8080 CPU through a 13 byte disk control block (DCB) located at a fixed location in memory.

Peripherals supported by the Sycor 440 system include synchronous and asynchronous communication devices, up to eight display terminals, serial and line printers, and card readers. The NPS configuration has four display terminals consisting of a typewriter-like keyboard and CRT display device. Each terminal displays a DMA image of a 576 byte terminal buffer located in RAM. Keyboard input is accomplished by software translation of a keyboard matrix code into the corresponding ASCII character code. For hardcopy output the NPS 440 includes a Centronix serial matrix printer.

Several different auxiliary storage devices may be attached to the Sycor 440 in addition to the mini-disk. These include magnetic tape drives, cassette tape drives, and floppy disk drives. The NPS configuration includes a cassette tape drive located in the control unit. This drive provides compatibility between the Sycor 440 system and the Model 340 debugger.

The Model 340 Communications Terminal is a complete system in its own right which is marketed by Sycor for remote job entry (RJE) applications [13]. When utilized as a hardware debugger, the 340 is augmented with 4K of RAM and a backplane coupling to a special interface board in the 440 control unit. The 340 debugger is provided with a software package which includes provisions for loading and dumping

hex format program files between cassette tape and 440 RAM, examination and modification of individual locations in 440 memory, inserting breakpoints and traps in programs executing on the 440, and single-stepping through a program executing one instruction at a time [15].

There are several hardware characteristics of the Sycor 440 system which strongly influenced the implementation of MTS. The most important of these are:

(1) 8080 CPU architecture

(2) terminal design

(3) mini-disk interface

(4) single-state CPU

(5) lack of memory protection

The impact which each characteristic had on the design and implementation of MTS is covered in chapters IV and V. For a more detailed discussion of Sycor 440 hardware characteristics see Ref. 1.

IV. MTS DESIGN

MTS was developed in order to integrate the Sycor 440 into the tutorial and research activities at NPS. The original problem definition given in section I.C provided general guidelines for accomplishing this objective. Restated briefly, they were:

(1) MTS was to support a suitably modified version of CP/M in order to provide compatibility with existing software development facilities.

(2) MTS was to utilize features of the Sycor operating system, notably the Sycor file system, wherever possible in order to reduce development time, avoid duplication of facilities, and allow the Sycor operating system to be used.

(3) MTS was to provide a man-machine interface at the terminals which was simple, flexible, convenient, and incorporated the best features of existing interactive systems at NPS.

The functions performed by an operating system may be divided into the four major categories of processor management, memory management, file management, and device management [10]. The general guidelines mentioned above effectively eliminated file management from consideration in the design of MTS. At the same time, the selection of CP/M as the operating system hosted by MTS provided a familiar,

well-defined model to use in determining the operational requirements of the system.

The following paragraphs describe the final design adopted for the Microcomputer Timeshared System from four different points of view corresponding to the four major interfaces which can be identified between MTS and its operating environment. The internal view covers the functional components of the system and their interdependencies. The external view deals with the relationship between MTS and the Sycor 440 operating system. The MTS environment as seen by a user program is described in the user program view. The last of the four, terminal user view, describes the interface between a terminal user and MTS, and the command processor which bridges this interface.

A. INTERNAL VIEW

1. VMM Concept

MTS was originally envisioned as a software interface between the bare Sycor 440 machine and up to four user tasks executing concurrently. In order to maintain compatibility between the Sycor 440/MTS system and existing NPS microcomputer development facilities, each user task would be provided with an operating environment identical to that found on the INTELLEC 8/MOD 80 system.

It was recognized that the Virtual Machine Monitor concept provided the simplest and most elegant means of

implementing such a software interface. A properly designed VMM could utilize the mini-disk included in the Sycor 440 hardware configuration to simulate the operation of multiple floppy disk drives and provide each user with a dedicated virtual printer through spooling. Since the INTELLEC 8 and Sycor 440 microprocessors are both based on the Intel 8080 CPU chip, the machine language instruction sets of the two are identical and interpretive execution would not be necessary. The virtual operating environment viewed by a user task would be the same as that found on the INTELLEC 8/MOD 80. Additionally, the monitor itself would be invisible to the user.

While the VMM concept provided the ideal solution in theory, there were several hardware limitations which precluded its implemention. The discussion of the VMM concept in chapter II lists four hardware characteristics of the host computer necessary to implement a practical virtual machine monitor. These four characteristics are essential in providing the reliability, integrity, and protection required in a timesharing system. Unfortunately, the Intel 8080 chip processors satisfy none of these requirements. The 8080 is a single state CPU. No distinction exists between system and user modes of operation; consequently, there is no need to provide both privileged and non-privileged instructions in the machine's instruction set. This makes it impossible on the 8080 to trap I/O instructions executed by a user task, and means that all I/O devices are accessible to whichever task currently controls

the CPU.

A more serious problem is the lack of memory protection. The 8080 provides neither bound registers nor address translation hardware to detect memory references outside the user task's address space. This shortcoming not only made the VMM concept impractical, but had a noticeable impact on the design finally adopted.

Perhaps the strongest argument for using the VMM concept in implementing a timesharing system is the transparency of the VMM's operation to user tasks. Since each user task is effectively running on an independent bare machine, the VMM imposes no restrictions on the user beyond those imposed by the machine's architecture. It was found necessary in the design of MTS to compensate for the limitations of the 8080 CPU by imposing certain restrictions on the freedom of user tasks.

2. MTS Monitor Concept

MTS was designed to provide a timeshared, virtual 8080 microprocessor environment for microcomputer systems development. The term "virtual" is appropriate here because the user actually interfaces with MTS for many services normally provided by the hardware in a dedicated CPU environment. A software interface between user programs and the Sycor 440 hardware was necessary in order to allocate the hardware resources equitably and efficiently, while at the same time satisfying the service requirements of several competing user tasks.

The design of the MTS interface drew heavily on the VMM concept. MTS was designed to multiplex only the physical resources of the Sycor 440 system, i.e. memory, the CPU, auxiliary storage on the mini-disk, terminals, and other peripheral I/O devices. No attempt was made to provide additional functional enhancements such as a file system or software development tools. As stated in the general guidelines for the project, it was expected that MTS would support a suitably modified version of the CP/M operating system, making such enhancements redundant.

The NPS Sycor 440 hardware configuration did not include floppy disk as an auxiliary storage medium. Since floppy disks are provided by the INTELLEC 8/MOD 80 systems, and are necessary to run CP/M, it was decided to implement virtual floppy disks and disk drives. Provisions were also made to expand the system to include virtual printers, card readers, paper-tape readers, and other dedicated I/O devices at a later date.

As a timesharing system, MTS is characterized by the following major features:

(1) the use of swapping to implement multiprogramming

(2) the use of interrupt driven processor management based on a round-robin scheduling algorithm

(3) the use of virtual floppy disks as the primary auxiliary storage medium

(4) the sharing of a single dedicated I/O device by multiple users.

Each of these characteristics is discussed in detail in this

and the following chapter.

3. Memory Management

The Sycor 440 provides 64K bytes of random access memory. Slightly over 3K bytes of this is required for terminal and cassette DMA buffers, a ROM bootstrap program, a mini-disk control block, and interrupt processing. The decision was made that a minimum of 48K contiguous bytes of memory would be made available to user tasks with the remaining 13K bytes reserved for the resident portions of MTS. This estimate of the amount of memory required by MTS was based in part on the size of the CP/M operating system and was intentionally generous to provide sufficient memory for future growth and enhancement of MTS facilities.

Once a decision had been made on the amount of memory available to user tasks it was necessary to select the optimum method of managing this memory space. This decision also was strongly influenced by hardware considerations. The Intel 8080 CPU provides only a single direct addressing mode. Since the Sycor 440 provides no address translation hardware, the more advanced memory management techniques such as paging and dynamic partitioning were infeasible. The only practical choices seemed to be swapping or a static partitioning scheme.

The static partition approach was considered carefully since it offered several advantages over swapping. Most importantly, a memory management technique which does not involve data transfers between memory and auxiliary

storage would be much faster than swapping. Static partitioning offered the added advantage of simplicity. With all user programs resident in memory it would not be necessary to maintain tables showing the disk addresses of blocked or ready tasks.

There was a single overriding factor which ultimately mandated the selection of swapping. A timesharing system must maintain the integrity of all concurrently executing tasks. Since the Sycor 440 provides no memory protection, this would be impossible if two or more tasks were resident in memory simultaneously. Not only would it be impossible to prevent one task from accessing another, it would be impossible to detect the occurrence of a reference out of bounds.

The use of swapping provides physical as well as logical separation of all user tasks in the system. Associated with each of the four terminals is a mini-disk file used to store a memory image of that terminal's current task when it is waiting for the CPU or blocked pending some I/O operation. At any given instant a task may reside on the mini-disk in its swap file or in memory, but at no time can two or more tasks be resident in memory simultaneously.

Swapping also makes available to each task the entire 48K user area of memory as originally planned. Even though few users will ever write a single program which requires the entire 48K bytes, the additional memory does enhance the operation of CP/M development tools such as the text editor by allowing larger buffers and fewer disk

accesses.

While the incorporation of swapping into MTS did solve the problem of maintaining task integrity, it also had the undesirable side effect of making the mini-disk data transfer rate the limiting factor in system responsiveness. Based on informal timing figures provided by Sycor, it was estimated that as long as six seconds may be required between timeslices to handle the swapping of two 48K byte tasks. This problem was recognized early in the design phase of development and solutions were sought in several areas. A partial solution to the problem is offered by an improved mini-disk controller under development at Sycor as this was written. Preliminary tests have shown that this improved controller will improve the mini-disk transfer rate by a factor of from three and one-half to four over its present value.

Several features have been included in the MTS design to minimize the delay inherent in swapping. For example, the system default size was limited to 16K bytes in an effort to reduce the size of the memory image which must be transferred to the mini-disk. A user can specify a larger memory size if required, but, in the absence of an explicit request for more memory, 16K bytes is assumed. Consideration was also given to minimizing the frequency of memory image transfers. The scheduler was designed to defer swapping until it determined that a different memory image was required to continue processing. As an illustration, consider the case of a single task active on the system. If

swapping is not deferred, this single task will constantly thrash back and forth between memory and the mini-disk. By deferring each swap until it is absolutely necessary, the overhead due to swapping becomes roughly proportional to the number of active tasks minus one, or zero for a single task.

4. Processor Management

An assumption basic to the entire MTS development effort was that MTS would support microcomputer systems development. This assumption implies that at any one time some of the tasks active on the system may require relatively large amounts of CPU time while others may be blocked pending terminal I/O. In order to ensure that the CPU resource is allocated fairly to all active tasks it was decided to implement an interrupt driven task scheduler. Each task is allocated the CPU for a fixed interval of time called a timeslice. A task retains control of the CPU until a hardware timer generates an interrupt signaling the expiration of the task's timeslice. The timer interrupt handler then transfers control to the task scheduler to select a new task for execution.

Notice that interrupt processing is carried out independently of task scheduling. This could cause a problem if a timer interrupt should occur while system code is being executed. The task currently residing in memory would be swapped out regardless of its status. This problem was resolved by designing MTS to set a software lock each time it is entered. The timer interrupt handler checks this

lock whenever a timeslice expires and suppresses swapping until the lockout condition has been cleared.

5. Virtual Floppy Disks

The MTS virtual floppy disk drive provides auxiliary storage for user programs on virtual floppy disks. These simulated hard-sectored disks have 128 bytes per sector, 26 sectors per track, and a maximum number of tracks determined by the size of the mini-disk file containing the disk image. Each user has eight drives available for dedicated use.

A virtual floppy disk resides on a block of logically contiguous mini-disk sectors. Where a physical floppy disk has a fixed capacity of 256K bytes, an MTS virtual disk may have any convenient size. MTS assumes that the disk image is made up of contiguous 128 byte floppy disk sectors starting with track 0 sector 1, proceeding through the 26 sectors of track 0 to track 1 sector 1, and so on until the virtual disk file is full.

In order to access a virtual floppy disk a user program must provide MTS with a complete sector address and the base address of a 128 byte DMA buffer in the user's memory space. This buffer derives its name from the nature of the data transfer operation: to the user program it appears that data is transferred between his program and the virtual floppy disk by direct memory access.

A complete sector address consists of a virtual drive number and the virtual disk sector and track numbers. With this information MTS can calculate the offset of the

addressed sector in the mini-disk file, validate that the addressed sector is within the file, and read the mini-disk sector into memory for transfer of the specified 128 bytes to the user's DMA buffer.

The mapping function used to convert a complete sector address into a mini-disk sector number is based on two facts:

(1) The virtual floppy disk image is stored as a linear array with the sector number increasing most rapidly and the track number increasing least rapidly.

(2) The mini-disk file containing the virtual disk image is a single contiguous block of 512 byte sectors.

Recognizing that the virtual disk image is stored as a linear array, the offset of any virtual disk sector in the file becomes

vs.offset = (26 * track.nr) + sector.nr.

Applying the fact that each 512 byte mini-disk sector may contain up to four virtual floppy disk sectors, the file offset and position of the specified virtual disk sector in the mini-disk buffer can now be calculated.

file.offset = vs.offset DIV 4

buf.pos = (vs.offset REM 4) * 128

where DIV and REM represent integer division and remainder respectively. Once the file offset is known, it is a simple matter to add this value to the file's base sector number to complete the mapping. Similarly, the buffer position is added to the mini-disk buffer's base address to find the current memory address of the specified virtual disk sector.

B. PROTECTION AND RECOVERY

Thus far in the discussion of MTS design features the topic of protection has already been encountered several times. It has been stated that the independence of concurrently executing processes must be maintained, and that protection is also necessary for the operating system itself. Due to the architecture of the Sycor 440 system, a great deal of consideration was given to the protection problem as well as the related topics of recovery and privacy.

1. System

MTS maintains the independence of concurrently executing user tasks by enforcing physical separation through swapping. This technique is practical and effective, but only deals with part of the protection problem. It is even more important that MTS itself be protected from user tasks. Situations in which a user task may inadvertently overlay or modify pieces of system code must be avoided. At best this could lead to a system crash with varying degrees of impact on all system users; at worst, MTS could continue processing in an erratic fashion which causes irrecoverable damage to user files. The first situation is immediately obvious to the system users; the second may go undetected for just long enough to be catastrophic.

The technique used to protect user tasks from one another cannot be applied to this new problem. While it is

feasible to make some sections of the system such as the task scheduler or terminal command processor non-resident and swap them in as required, there are other system functions such as interrupt handlers which must be resident at all times. One possible solution to the problem would be storing all system code in read only memory with backup copies of all system data maintained on the mini-disk. The simplicity of this approach makes it appealing, but nevertheless impractical for reasons discussed in the external view of MTS presented in section IV.C.1.

After a great deal of consideration, no practical, effective method of providing system protection could be found which did not involve imposing an unacceptable amount of protection overhead on the system. In lieu of a protection scheme it was decided to incorporate a recovery capability into the MTS design. It was felt that the ability to recover gracefully from occasional inadvertent system crashes would compensate somewhat for the lack of adequate system protection.

Three different types of information were identified as being necessary to the implementation of a recovery capability. They were:

(1) a copy of each user task's latest memory image,

(2) copies of all current values for system control variables and tables,

(3) the identity of the task running when the crash occurred.

The use of swapping made the first type of information

readily available on the mini-disk with no additional action required. To this point in the design it had not been found necessary to save the values of system control variables and tables anywhere other than in memory. This deficiency was rectified by introducing the concept of a system state block (SSB).

The SSB consists of a compact, contiguous data block in the system area of memory containing all control data defining the state of the system at each instant. The SSB includes information such as the status of each user task, virtual floppy disk assignments, the location of each swap file on the mini-disk, the identity of the task currently allocated the CPU, and the protection attributes and locations of all virtual disk files. In order to have this information available should recovery be required, the SSB is copied to the recovery file whenever the state of the system changes.

Taken together, the four swap files and the recovery file contain sufficient information to restore normal execution after a system crash. The third item required for recovery, the identity of the task causing the crash, is needed to delete the faulty task from the system. The swap file for this task contains a duplicate of the memory image which has just caused the system failure. If this same task were reloaded, another system crash would likely occur.

One final point must be mentioned to complete the discussion of system protection, and that is access protection. Most general purpose timesharing systems employ

some type of code to identify individual or group users of the system. This code may be used in the collection of utilization or accounting statistics, or simply to identify authorized users of the system. It was decided that the access protection provided by such a code is not a necessary requirement for MTS. The NPS microcomputer laboratory has generally followed an open door policy which permits anyone with an interest in microcomputers to use the available facilities. Conforming to the spirit of this open lab policy, access protection is not implemented in MTS.

2. Virtual Floppy Disks

The protection issue surfaced for the third time in the implementation of virtual floppy disks. In this context, protection implies privacy and security for virtual floppy disk files.

From the viewpoint of privacy and security of data, floppy disks provide an auxiliary storage medium surpassed by few others. The physical disk is small in size and convenient to carry or store, yet provides a storage capacity of 256 kilobytes, equivalent to 1.6 card sleeves. When an individual finishes an operating session on a system equipped with floppy disk drives, he need only physically remove the disk from the drive to provide whatever physical protection the contents merit.

The virtual floppy disks provided by MTS do not afford the same degree of privacy or security as a physical disk. With the Sycor 440 hardware configuration which

existed when this was written, it was not possible to copy a virtual floppy disk image onto a physical disk for safekeeping.

In order to provide a limited degree of protection for virtual floppy disks it was decided to provide three different protection attributes for virtual floppy disk files:

(1) no protection - unlimited access for read or write

(2) restricted - unlimited access for read only; write access only if the file's protection key is known

(3) protected - read and write access only if the file's protection key is known.

The first option, no protection, applies to scratch disks provided for temporary storage or work space. Such files should be available to any user as required. The restricted attribute allows the owner of a given virtual disk file to make the contents of the file available to others on a read only basis. This level of protection would apply to the system disk containing the CP/M operating system. The protected attribute provides complete protection for a virtual disk file. No one other than the file's owner or someone designated by the owner can access the file either to read or write.

Since MTS does not require user identification codes, the protection key feature was adopted to identify the owner of a virtual disk file. A protection key may be any combination of zero to four ASCII characters associated with a virtual disk file when the restricted or protected

attribute is set. The same protection key must be / supplied by a user attempting to write to a restricted file or read and write to a protected file.

C. EXTERNAL VIEW

When the Sycor 440 Clustered Terminal Processing System was delivered to the NPS microcomputer laboratory, the shipment included an extensive package of software in addition to the hardware listed in chapter III. The existence of this software had a significant impact on the design of MTS. The nature and extent of this impact is discussed in the following paragraphs.

1. Sycor 440/MTS Interface

The Sycor 440 system demonstrates one way in which microcomputer technology has been applied to commercial data processing. The 440 system is marketed for applications in the field of data entry, in particular data entry at locations remote from a central processing facility. This intended application is readily apparent in the software provided by Sycor - formatted and unformatted keyboard input, remote job entry, report generation, communications, and on-site file enquiry are prominently featured. The 440 operating system and file management facilities also reflect the emphasis on data entry. Most of the programming aids which are important in a system development environment are not provided by the Sycor 440 software package. There is no text editor, 8080 assembler, debugging program, or 8080 hex

format loader. The operating system is large, allowing only limited memory space for user developed programs. Further, the mini-disk file system is oriented around static allocation of fixed size files.

The Sycor 440 operating system and associated utility programs are well suited to the data entry application for which they were designed. At the same time, they do not provide a suitable environment for microcomputer systems development. This fact provided the initial motivation for the development of MTS.

Once the decision had been reached to design and implement a separate system to support systems development on the Sycor 440 hardware there were basically two options open with regard to the Sycor supplied software:

(1) remove the Sycor operating system and associated disk files to make more memory and disk space available for MTS

(2) design MTS so that both systems might be available concurrently.

Option 2 had the desirable effect of making available a resident COBOL compiler and a PLMR cross-compiler, both of which are supported by the Sycor operating system. Option 1 allowed the greater flexibility in the design of MTS, but reduced somewhat the overall capabilities of the Sycor 440/MTS combination. The deciding factor in selecting the second option for implementation was a desire to furnish the system affording the maximum potential for future development while still meeting project goals.

2. File System

The decision to make the Sycor operating system and MTS concurrently available made it possible to implement MTS without designing a file system. The file management facilities provided by the Sycor operating system proved adequate to handle swapping, recovery, and virtualized floppy disks. Constraining each file to a predetermined fixed size was not difficult; in fact, this feature of the Sycor file system simplified considerably the virtual disk mapping function on which the entire virtual floppy disk design was based.

Four different types of files were included in the final MTS design. They are:

(1) swap files - four files, one of which is associated with each of the four terminals, containing a user program memory image

(2) recovery file - a single-sector file containing a current copy of the system state block

(3) configuration file - a single-sector file containing the user defined name and protection attribute for each virtual floppy disk

(4) virtual floppy disk files - a maximum of 32 different files, each containing the image of a floppy disk as described in section IV.A.5.

Three of these four file types have been encountered before, but the configuration file type requires some additional explanation. MTS identifies virtual disks by a logical disk number in the range 0 to 31. Since each virtual floppy disk

actually resides in a mini-disk file created under the Sycor operating system, there must be some mechanism for mapping a logical disk number into a file name contained in the mini-disk directory. There is an additional requirement that the protection attributes assigned to various virtual disk files be saved between MTS operating sessions. Both of these functions are performed by the configuration file.

The configuration file is made up of 32 entries of thirteen bytes each contained on a single mini-disk sector. Each entry is composed of a zero to eight byte filename, a zero to four byte protection key, and a single byte protection attribute. The logical disk number for each entry is simply the position of that entry within the file.

In order to reduce the number of mini-disk accesses required to service virtual floppy disk I/O requests, it was decided to read the configuration file into memory only once, during MTS initialization, and abstract the contents into an internal data structure resident in memory. This concept was expanded later to include the mini-disk file directory as well. That too is read into memory during initialization and searched for all four MTS file types.

D. USER PROGRAM VIEW

The MTS user program interface consists of a set of service routines which may be called by a user program through a single entry point to perform terminal I/O, access virtual floppy disks, or modify the user's virtual

environment. The design was heavily influenced by the CP/M operating system which uses a similar scheme for I/O [2]. The use of explicit calls to MTS for I/O was an undesirable limitation necessitated by the architecture of the 8080 CPU. Since the 8080 is a single-state machine without privileged instructions, the hardware provides no facilities for trapping I/O instructions.

A call to MTS takes the form

<value> = MTS(<fid>,<parm>).

The first argument, <fid>, is a number which identifies the function to MTS. The <parm> argument may be a parameter value, if only a single parameter is required, or the address of a parameter list if more than one is needed. In each case, MTS returns <value> upon completion of the requested operation. This returned value may be an ASCII character code, an error code, or zero if the value has no significance.

The services available to a user may be logically divided into two types of calls on MTS: (1) system calls, and (2) service calls. System calls handle creating, modifying, and deleting attributes of a user's virtual environment. As a minimum, each user is provided with a virtual system consisting of an 8080 CPU, 16K bytes of RAM, a terminal, and a single floppy disk drive with a read-only copy of the CP/M operating system loaded. This configuration is the system default environment assumed when the user logs in. Beyond that the user may request:

(1) additional memory up to a maximum of 48K bytes

(2) additional floppy disk drives up to a maximum of 8

(3) attachment of any virtual floppy disk whose protection key is known

(4) reboot of the user's system from drive A.

Each system call performs the same function for a user program as the corresponding terminal system command performs for the user at a terminal.

Service calls provide a user program with access to the terminal, virtual printer, and virtual floppy disks. The details of the terminal interface are discussed more fully in section IV.F. In the context of service calls, it is important to note that attempting to read from a terminal when no input is waiting will cause the requesting task to be swapped out and blocked until input is available.

Before a user program can access a particular virtual disk, the user must attach that disk to one of the eight virtual disk drives, either by making a system call to MTS, or entering the proper system command at the terminal. Once a virtual disk has been attached, any sector within that disk is accessed by specifying a complete sector address and a DMA buffer address. MTS provides a service call to enter each of the three components of a complete sector address or the DMA buffer base address as well as additional calls to read or write the specified sector.

In the section which discusses the VMM concept it was mentioned that transparency to the user is an important characteristic of a VMM. The attempt was made in the design of MTS to incorporate this same feature. Due to the

hardware constraints imposed by the 8080 CPU, this attempt was not entirely successful. There are several places in the user program/MTS interface where the operation of MTS could not be hidden. The necessity for explicit calls to MTS for I/O services has already been mentioned. The manner in which interrupts are handled by the 8080 also impacts on a user program. Since the only addressing mode provided by the 8080 is direct addressing, the fact that MTS and a user's program occupy the same physical memory is apparent to a user program by the limits on the user's address space.

E. TERMINAL USER VIEW

The primary interface between the terminal user and MTS is provided by a command language interpreter called the MTS Command Processor (MCP). This interface provides the user with the facilities necessary to establish and modify the working environment. The user can gain access to MTS through system commands at any time, even though currently communicating with a subsystem such as CP/M or other programs. Access is accomplished by entering the appropriate command at the terminal, terminated by the ERROR RESET key. This signals the MTS monitor that the input data is a system command to be processed by MCP.

1. MTS/MCP Interface Design

One of the design consideratons for a command processor in an interactive timeshared environment is whether it shall be resident or non-resident. The actions

required to establish and modify the user's environment are not normally time sensitive nor continuously exercised by the user. For these reasons, recent timesharing systems have commonly implemented the command processor as a non-resident task [16]. Establishing MCP as a non-resident, swappable task was given serious consideration. However, one of the major Sycor 440 hardware limitations affecting the implementation of MTS was the relatively low data transfer rate between memory and the mini-disk (I.A.3). To meet the goal of reasonable system response, minimizing the number of swapping tasks became one of the MTS design constraints.

The decision was made to design MCP as a resident but completely independent module of MTS. This concept makes the MCP/MTS interface essentially the same as that between any users program and MTS. MCP affects changes in the user's virtual environment by calls to MTS in the same manner as would a user program. There are two primary differences between the MCP/MTS interface and a user program/MTS interface:

(1) The entry port used by MCP is an internal MTS entry point. MTS also provides an external entry point for use by user programs. The requirement for two entry points resulted from the decision to make the MCP a resident process. The distinction is necessary to bypass the mechanism which blocks user programs requesting terminal input when the input buffer is empty.

(2) MCP must save and restore the MTS system stack pointer to ensure returning to the proper location in the MTS monitor. User programs do not directly interface with the MTS monitor, and thus are not concerned with saving and restoring its stack pointer.

The independence of data structures and system call interface features were maintained for ease of implementing MCP as a swappable image at some future time. Upgrading the Sycor 440 mini-disk controller speed and an increase in the number of system commands available to the user would justify implementing the MCP as a non-resident swappable task.

2. System Commands

Since the system command language is the terminal users main point of contact with MTS, the features and syntax were given careful consideration. The design goals were to develop a command language which is easy to learn and easy to use.

There are two basic sets of system commands available to the user. One set of commands allow the user to establish and modify the environment. These include linking the terminal to MTS (LOGIN); specifying virtual disk drives and virtual floppy disks to be attached to the environment (ATTACH); changing the memory image swap size allocation (SIZE); and unlinking from MTS (QUIT).

The second set of commands provide the user with a means of specifying protection attributes for virtual floppy

disk files. These include adding the protection attribute to a specified virtual floppy disk (PROTECT); adding the restricted attribute to a protected virtual floppy disk to allow read access by other users (RESTRICT); and to remove all previous protection attributes from a virtual floppy disk (UNPROTECT).

The function, syntax, parameters, description and associated error messages for each command are described in detail in section D.3 of Appendix A.

F. TERMINAL INTERFACE DESIGN

The four terminals attached to the Sycor 440 system provide the user with a CRT display and typewriter-like keyboard for entering data. A brief description of the terminal hardware is contained in chapter III. The factors affecting the terminal interface design included:

(1) Long delays between key depression and appearance of a character on the terminal display are unacceptable to the user.

(2) The characters displayed at each terminal are a DMA image of characters located in the Sycor 440 main memory.

(3) The terminals have very primitive hardware display logic. Thus software is required to accomplish such tasks as converting a keyboard matrix code to ASCII code for each key depression, blinking the current position indicator (cursor), and scrolling the

display.

1. Design Decisions

MTS was designed to provide a virtual terminal interface for programs requesting terminal I/O. This interface simulates the operation of a serial half-duplex console device. For input of data, the user program requests characters one at a time through service calls to MTS. If input is available, the next character is returned to the requesting program. Output is handled in a similar fashion with the user program providing an ASCII character code as an argument in the appropriate service call. A service call is made for each character to be displayed. For each request received by MTS, the character is displayed by placing it at the current cursor position in the appropriate terminal's DMA display buffer. The only characters not placed in the display buffer on output are: carriage return (CR), which returns the cursor to the leftmost position of the current line; and line feed (LF), which moves the cursor position down one line. The output of a data sequence is normally terminated by a CR/LF character combination. MTS also provides a terminal status service call which allows a user program to test whether input data is available for processing (D.3). This function could be used to test for a break key indication during output by the user program.

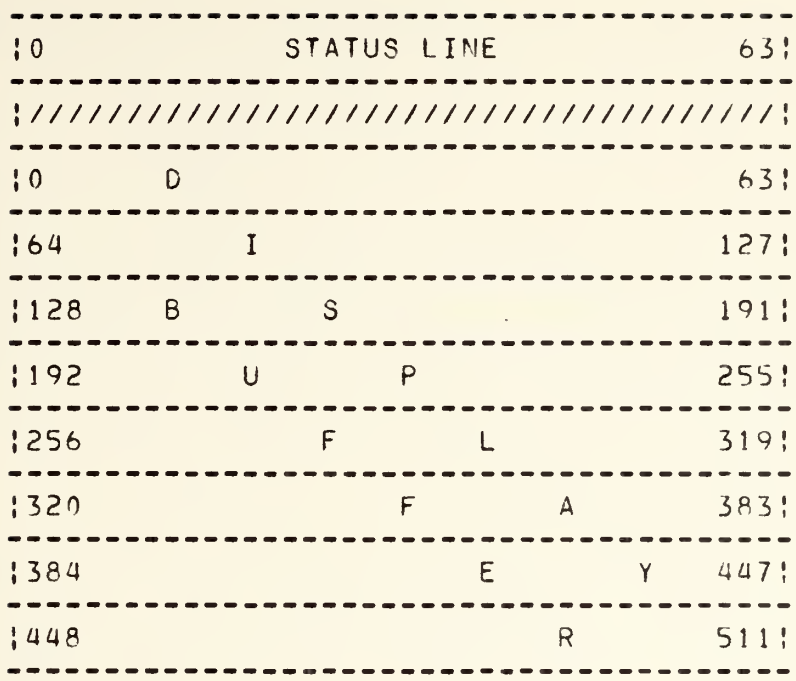
Due to the unavoidable delays in user program response caused by swapping and aggravated by the relatively

low data transfer rate of the mini-disk, the MTS terminal interface was designed to provide character echoing and simple line editing. This ensured reasonable response times to key activation by the user, even though his program was not currently in memory. Each character key activated at a keyboard results in the appropriate character being displayed on the CRT by MTS. The line editing features are discussed in section 3 below.

There are four 576 byte DMA terminal buffer areas located in RAM. Thus, a 2304 byte area of the Sycor 440's main memory has been allocated by the system hardware design for terminal display buffers. To take maximum advantage of this memory utilization, the decision was made to use this area as the input holding buffer as well as the display buffer. This eliminated the need for additional input buffers and the extra processing time required to move a completed input line to a separate buffer.

2. Display Description

The first 64 display character positions of each terminal are physically separated from the remaining 512 character positions by a blank line. This led to the decision to reserve the first line for display of status and error messages. All input and output of data was to be accomplished in the remaining 512 character positions. Thus, each 576 byte terminal buffer is logically divided into two separate buffers, as follows:



The numbers are decimal and specify character positions within the status line and display buffer.

The terminal status line is used by MTS to display three types of status information:

(1) The current virtual drive and floppy disk assignments for that terminal.

(2) The size of the user's swap image, i.e. the amount of memory space currently available.

(3) Error message alerts produced by MTS system commands, or resulting from user program calls on the DISPLAY MSG service routine (see section F.2 of Appendix A).

The status line display format and contents are discussed in detail in section E of Appendix A.

As indicated by the above format description, the display buffer can hold a maximum of 512 characters (8 lines on the CRT). As previously mentioned, this buffer not only

provides the display of characters but also acts as an input buffer, holding the input data until the user's program requests it. Since MTS provides simple line editing of input data, this data can not be considered available to the user's program until an input line termination character has been received by MTS. To establish an input buffer for a program, the user enters the data and terminates the line by hitting the NEWLINE or ENTER keys on the keyboard. This establishes that line as an input buffer available for processing by the user's program. Note that the key combinations 'I/O CTL M' or 'SHIFT CR' (on the number pad) will also result in the termination of an input line. Either of these keys, as well as NEWLINE and ENTER, may be used for line termination.

Once an input buffer has been established the user may continue to input data on the next line. The user may use any of the line editing or other cursor control features on this new line of input data. However, this new line may not be terminated until the user's program has processed the previous input buffer (see terminal alerts below).

Each character output from the user's program is displayed at the current cursor position. Each output results in all input buffer pointers being reset to the character position at the end of the output data. Thus, subsequent I/O will start at this point. This implies that if the user had been in the middle of entering data when the output occurred, it must be reentered.

The MTS terminal interface provides the user with

either a visual or audio response to each key depression. Normal visual response is provided by display of the entered character (echoing) and/or movement of the display cursor. The display cursor is a blinking underscore character which marks the current position on the screen. Data is always entered and displayed at the current cursor position.

The audio responses consist of either a beep or click at the terminal. A terminal beep alert will be generated for any of the following conditions:

(1) An input buffer is waiting to be processed by the user's program and the terminal user attempts to terminate a new input line.

(2) An attempt is made to move the cursor back past the start of the current line. For example, attempting to delete the previous line or character after the line has been entered by a termination key will result in a beep.

The terminal click alert is associated with the display scrolling feature. Since the display buffer also acts as an input buffer, scrolling the display when the 512 byte display buffer is full could destroy input data which has not yet been processed. For example, the user could be entering a 512 character string. Upon termination of that input line, MTS will prevent scrolling until the user's program has processed the first 64 characters. This ensures that the input data is not destroyed by the scrolling operation. This scrolling lockout is indicated to the user

by a terminal click alert.

3. Terminal Key Functions

The terminal keys fall into five basic functional groups:

(1) Character string keys - these are the alphanumeric and special character keys normally available to the user for input of data. Alphabetic characters are entered and displayed in either upper or lower case, depending on the key mode (see below). The tab key causes the entry and display of a special tab character, no blanks are padded in for tabs.

(2) Entry mode keys - these keys define the interpretation of keys for special functions or alphabetic upper case. These include the FUNCTION SELECT, I/O CONTROL, and SHIFT keys. In addition, the key combination FUNCTION SELECT and C sets or clears the alphabetic key entry mode to upper or lower case. This functions as a shift key lock because a physical lock is not provided with the terminal keyboard.

(3) Line termination keys - these keys define the termination of an input line of data. Input data to be processed by the user's program must be terminated with one of the following keys or key combinations: NEWLINE; ENTER; I/O CTL M; or SHIFT CR. Their function was discussed in the preceeding section. Another termination key is ERROR RESET, which specifies that the input line it terminates is to be

processed by MTS as a system command.

(4) Line editing and cursor control keys - these keys provide the simple line editing features such as line delete (NEXT FMAT); character delete (BACKSPACE); clear screen (FS $ or I/O CTL $); and move cursor left or right (<-- ; -->).

(5) Number pad keys - consists of 10 numeric digits and 8 ASCII control characters located on the right side of the keyboard. The digits function in the same manner as the other numeric digits on the keyboard. The ASCII control characters are displayed when the SHIFT key is depressed in conjunction with the appropriate key. The only control character which affects the display is SHIFT CR (see line termination keys).

All terminal keys and their functions are described in more detail in section D.2 of Appendix A.

G. CHOICE OF A PROGRAMMING LANGUAGE

Following the design phase, it was necessary to choose an appropriate programming language to be used in the implementation phase of development. An obvious requirement was that the language selected must support system programming for the 8080 microprocessor. Other factors involved in choosing a language for the MTS development included the following:

(1) Efficiency of the code generated by the assembler or compiler. An important consideration in any operating

system development is minimizing the amount of memory and processing time utilized by system routines.

(2) Ease of access to machine resources (e.g. registers, memory, I/O ports, stack, etc.) through the constructs provided by the language. This must be considered whenever developing software which will be interfacing with the bare machine.

(3) The availability of the assembler or compiler for development work. The turn-around time for assembly or compile tasks and the debug aids provided by a language are important factors.

(4) The inclusion in the language of convenient control structures and self-documentation features similar to those found in most high-level structured languages. It has been shown that these features assist in rapid system development and checkout, straightforward maintenance and modification, and a highly reliable system. It was envisioned that MTS would be enhanced and modified extensively as the requirement for additional microcomputer development facilities increased. Thus, ease of future modification and maintenance was a prime consideration.

(5) The ability of the assembler or compiler to generate error free code was important for ease of coding, testing, and debugging the system modules during project development.

There were three languages available at NPS which supported programming for the 8080 microprocessor. The following is a brief description of these languages and their advantages and disadvantages:

(1) 8080 Assembly Language [8] - the assembly language developed for use with the 8080 microprocessor. It satisfied (1) and (2) above by providing efficient and direct access to the machine resources. Item (3) was satisfied because a resident assembler and dynamic debugging tool were available on an INTELLEC 8/MOD 80 microcomputer system for use during the project development. The major limitation in using an assembly language is its failure to satisfy item (4).

(2) ML80 [12] - a structured system programming language for the 8080 microprocessor developed as a thesis project at NPS. It is composed of two independent languages: M80 - a macro oriented language; and L80 - a machine oriented language. It incorporates such features as: control structures, similar to those found in most high-level structured languages; allows full use and control of the resources of the 8080 microprocessor through the use of algebraic notation for machine-level register and data operations; provides compile-time features, such as expression evaluation, conditional compilation, and macros; and provides load-time facilities, such as the linking of precompiled procedures into the object program (generates relocatable code). It was also available

as a resident compiler on the INTELLEC 8/MOD 80. Thus to some degree, ML80 satisfied all the considerations mentioned above except (5). That is, it had never been used for a large system development and thus its performance in that environment was untested.

(3) PL/M [9] - a high-level programming language designed to provide system programming for the 8080 microprocessor. It was designed to facilitate the use of modern techniques in structured programming. Thus, it came the closest in satisfying consideration (4) above. However, its major disadvantage was in generating less efficient code and allowing less direct access to the machine resources than the preceeding two languages.

After considering the advantages and disadvantages of the languages available, the decision was made to utilize ML80 as the primary development language for MTS. A subtask of the thesis project was to test ML80 as a programming language in a large system development environment. Results of using ML80, including recommended enhancements to the language and it's facilities, are included in the remaining chapters.

V. MTS IMPLEMENTATION

Top down system design and modular programming are current software engineering concepts which strongly influenced the design and implementation of MTS. As each specific functional requirement of a timeshared microcomputer system was identified, a new module was added to the MTS design to satisfy the requirement. An effort was made to make each logical module an independent entity communicating with other system modules through a simple, well-defined interface. This same concept was also applied to the implementation phase of development. Each logical module was implemented as a separate code module with the fewest possible number of intermodule linkages. Five modules were needed to meet the processing and resource sharing requirements imposed by timesharing. In addition to these code modules, a sixth declaration module was included to define the underlying data structure.

Use of ML80 as the implementation language for MTS affected the development primarily in the area of module and data structure linkages. The problems encountered can be attributed mainly to the limited memory size (16K) of the INTELLEC 8/MOD 80 which hosted the ML80 compiler. Due to the relatively small amount of work area available, many of the compiler's tables and stacks were too small to satisfy the demands of MTS. This size limitation forced a corresponding

limitation on the size of individual MTS modules and submodules. The effect of forcing a decrease in the natural size of the MTS program modules was to increase the linkage requirements between these modules. The ML80 compiler provides some link editing facilities, but does not conveniently support the numerous linking requirements encountered in the development of MTS.

A. SYSTEM STATE BLOCK

The management functions performed by MTS may be divided into task management and resource management. Task management is concerned with the control and record-keeping functions necessary to support up to four concurrent tasks. Resource management provides the same functions for the hardware resources of the Sycor 440. Both types of management involve recording status information for later use in processing control. Considered jointly, this status information defines the state of the MTS system and forms the basis of the MTS data structure.

In the implementation of MTS it was found necessary to combine all status information into a logical and physical block called the system state block (SSB). This approach was originally adopted to reduce the overhead of the recovery feature. The status information contained in the SSB must be copied to a recovery file after each swap. By consolidating this data into a compact, contiguous block, only a single mini-disk access was required.

Subsequently, it was found desirable to combine all variables referenced by two or more MTS modules into a single declaration module with global scope. This improved the readability of the source program, simplified program debugging, and facilitated maintenance of the system. The SSB is logically composed of three distinct elements:

(1) task control table - contains information on the state of each task and data required to support swapping. Each variable contains four entries - one for each of the four terminal tasks.

(2) disk map table - contains information on the status, protection attributes, and mini-disk location of all virtual floppy disks. Each variable contains 32 entries - one for each of the 32 possible virtual floppy disks.

(3) system control variables - single variables referenced by several modules, e.g. the swap lock and the number of the task currently executing.

Each task control table and disk map table entry was actually implemented as a named byte vector indexed by the task number.

B. MONITOR MODULE

The monitor module incorporates all of the task management functions required by MTS into a physically and logically distinct module. Included in this module are routines which handle the loading of a task, scheduling a

task for execution, swapping tasks between memory and the mini-disk, and deleting a task after an irrecoverable hardware error. Due to the module size limitation imposed by the 16K ML80 compiler, it was necessary to divide the monitor into utility and task management submodules for compilation. A third submodule, containing the initial program load routines, was included in the physical monitor module for reasons of convenience rather than logical continuity.

1. Utility Submodule

It was found convenient in the implementation of MTS to develop a group of utility routines to handle recurring primitive operations required to access entries in the system state block. Since most SSB entries are indexed by either a task number or a virtual disk number, a significant savings in memory space was realized by replacing in-line code segments with calls to a utility for indexing operations. The convenience of having these primitives available was offset somewhat by the necessity to compile the utilities submodule independently of the calling routines.

2. Task Management Submodule

The life cycle of a task in a multiprogrammed environment may be represented by a series of transitions between process states. It is the job of the operating system to control and monitor these transitions [10]. In the MTS operating system this function is performed by the

task management submodule.

Each user task in the MTS environment may exist in one of the following states:

(1) logged in - task is active but not yet loaded

(2) hold - task requires the services of the terminal command processor to alter its virtual environment

(3) blocked - task is waiting for the completion of an I/O operation

(4) ready - task is waiting for the CPU to become available

(5) running - task has been allocated both memory and the CPU

The current state of a task is determined by the bit pattern set in the status byte associated with that task. Each bit of this status byte corresponds to a different process state, and there is a different status byte associated with each task. Since the information contained in the task status bytes is an important part of the overall system status, they are located in the system state block.

The processing performed by the MTS monitor is driven by the values of the task status bytes. When the monitor is entered it sets a task counter to the number of the task currently running, increments the counter, and then examines each bit of the status byte indexed by this counter. If it finds a bit set, it calls the appropriate routine to affect the state transition implied by the task's current state. In the event no bits are set, the task counter is again incremented and the process repeated.

Note that the task status byte plays three different roles in the management of user tasks:

(1) It drives the operation of the monitor.

(2) It provides the simplest possible interface between the monitor and the terminal command processor. The MCP need only set the appropriate bit to inform the monitor that a new task has entered the system.

(3) It allows the terminal interface module to inform the monitor that a system command has been entered without interrupting the current task. This feature was implemented to reduce the swapping overhead of system command processing while still allowing MCP to be a swappable task.

Implicit in the sequential examination of bits carried out by the monitor is a hierarchy of process states. Since both the terminal interface module and the MCP, as well as the monitor itself, may set various bits in the status byte independently of each other, some method of resolving conflicts was necessary.

3. Initial Program Load Submodule

While not logically a part of the task management function, the IPL submodule has an important influence on the operation of the monitor after MTS is loaded. The preceding section described how the SSB, particularly the task status byte, drives the operation of the monitor. The primary task of the IPL submodule is loading values into the SSB before MTS is executed. Two loading options have been

implemented: initialization and recovery.

The recovery option is the simplest and most straightforward of the two. As discussed in section IV.B.1 the four swap files and the recovery file together contain all the information MTS needs to recover after a system failure. IPL processing is limited to searching the Sycor file directory for the recovery file's sector address and then reading the file into the SSB.

A great deal more processing is required to initialize MTS. Again, the Sycor file directory must be searched - this time for all four swap files, the configuration file, and the recovery file. Assuming that the six MTS system files are found, the IPL submodule then reads the configuration file and searches the Sycor file directory for each virtual floppy disk file. For every file found in the directory, system or virtual floppy disk, beginning and ending sector numbers must be recorded in the TCT and DMT respectively. Initialization of the DMT also requires that protection keys and attributes must be copied from the configuration file.

The IPL submodule provides the only internal interface between MTS and the Sycor operating system, i.e. the file directory. This submodule is also the only routine, other than the timer handler in the interrupt module, which calls the monitor. The final distinguishing feature of the IPL submodule is that it is the only non-resident module of MTS. Since IPL is only required once, immediately after MTS is loaded, the code was written to be

loaded in the user area of memory. Once normal operation begins, the IPL submodule is overlayed by the first user task which is loaded.

C. INTERRUPT MODULE

The Sycor 440 provides a comprehensive set of prioritized interrupts. Each hardware interrupt causes the execution of an 8080 RESTART 1 (RST 1) instruction. This instruction behaves like a call to location 0008H. That is, it results in the program counter value being pushed onto the current stack and control being transferred to location 0008H. Due to this hardware characteristic, user programs must ensure that any user defined stacks are at least four bytes larger than the maximum size required by the user's own code. This will ensure that the stacks will not overflow if their program is executing when an interrupt is generated.

Since all interrupts cause execution of the same interrupt instruction, some means must be available to determine which device generated the interrupt. The Sycor 440 solves this problem by defining an interrupt level for each different device. There are 17 interrupt levels with values ranging from 0 to 16. The priority of interrupts is established by assigning a higher priority to a device with a higher numeric value. When an interrupt occurs the level used to identify the initiating device is available on a specific input port. Simultaneous interrupts will be placed

on the interrupt input port in a priority sequence. That is, the levels will be input in descending sequence by level number. When the level on the input port is zero all pending interrupts have been processed. The interrupt input port number and the interrupt level assignments which apply to the current NPS Sycor 440 hardware configuration are contained in Ref. 1.

The interrupt module processes two basic interrupt categories:

(1) timer interrupt - this interrupt is generated by an internal clock once every 50ms, when it has been enabled. This provides MTS with the capability of accomplishing tasks which must be done on a periodic basis. It also provides MTS with the means for regaining control of the system from a user program.

(2) device interrupts - these interrupts are generated by a specific peripheral device to indicate the completion of a task or a request for service. The processing of these interrupts is device dependent.

1. Data Structures

The interrupt module consists of two parts, an interrupt controller and a set of interrupt handlers. There is an interrupt handling routine for each interrupt level processed by MTS. The data structures required by the interrupt module include:

(1) interrupt stack - a 30 byte data vector used to save the current system environment prior to any interrupt

processing and to restore the environment upon completion of interrupt processing.

(2) blink timer - a single byte used to store the current value of the periodic terminal processing counter. This counter is decremented once every 50ms and is used to determine when to blink the cursor character at each terminal.

(3) task timer - a global system variable used to control a user task's timeslice.

(4) lock - a global system variable used to indicate swapping lockout.

2. Interrupt Processing

During MTS initialization, locations 0008-000Ah are loaded with a jump instruction to the interrupt controller. Thus, when an interrupt occurs, the interrupt controller is called. It saves the current environment, identifies the interrupt level and calls the appropriate handler routine. Upon return from the interrupt handler, the controller checks the interrupt input port to determine if any more interrupts are pending. If so, the sequence is repeated; otherwise the environment is restored, interrupts enabled, and control returned to the interrupted process.

The interrupt handlers provide the following processing:

(1) timer handler - manages the two functions of MTS which occur at periodic intervals. These are blinking the terminal cursors and returning control to the system

when a task's timeslice expires. In order to keep track of the two intervals involved, two counters are maintained. These counters are each set to an initial value and then decremented each time a timer interrupt occurs. The actual value contained in either counter represents the time remaining in the interval in multiples of 50ms. When the blink timer reaches zero the appropriate routine is called to blink the cursor character at each terminal. When the task timer is zero a check is made to see if swapping is locked out. If not, the current environment is moved to the swap stack and control is transferred to the MTS monitor.

(2) terminal handler - processes the interrupt generated by a character key depression at one of the four terminals. It retrieves the terminal identity responsible for generating the interrupt and the associated keyboard matrix code from the appropriate input port. The terminal input control routine is then called to complete the key processing (see Terminal Interface Module).

(3) other device handlers - the remaining device handlers (e.g., cassette and printer handlers) were designed but not implemented. The present implementation simply clears the interrupt status and returns.

D. SERVICE MODULE

The most visible point of contact between MTS and user programs is the software interface which multiplexes the hardware resources of the Sycor 440. In the implementation of MTS all those procedures and routines which provide this resource management were consolidated into a single code segment called the service module. Once again, due to the module size limitation imposed by the ML80 compiler, it was necessary to divide the service module into three physical submodules.

1. User Interface Submodule

Section IV.D describes in detail the form of a call to the MTS service module. The implementation of this calling protocal follows the PL/M convention for parameter passing. That is, the function identifier (a single byte) is passed in register C, while the parameter list address is passed in register pair DE. The service module subsequently uses the A register to pass the return value back to the calling program. A user program accesses the service module by loading the C,D, and E registers with the appropriate values and then executing a call to location 2000H.

The service module provides an alternate entry point for service requests from other MTS routines. This alternate entry point was found necessary in order to implement the MCP as a swappable image. Since a second entry point was required, it was decided to make the interface as general as possible by using negative function

identifiers for internal service calls. This made it relatively simple to verify the source of the service request, and, as an added benefit, made available an entire new series of function identifiers without disrupting the series already in use for external calls.

The first action taken by the interface submodule upon entry is setting the software lock which suppresses swapping. The routine continues with validation of the function identifier and a call to one of the other submodules to handle the operation requested. When control is regained by the interface submodule the swap lock is reset and control returned to the calling routine.

2. Service Call Submodule

The first step in the implementation of the virtual I/O functions was to determine the operations performed by each device. Since it had already been decided to limit the scope of this baseline system, only three devices were considered: terminals, the printer, and virtual floppy disks sharing the mini-disk. A total of ten operations were identified as necessary to provide virtual I/O with these three devices. Subsequent efforts concentrated on the terminals and virtual floppy disks. Data structures were developed to support each device and routines written to simulate the operations performed by each. To conform with the modular approach adopted for this project, the terminal routines were incorporated in the terminal interface module. The service call submodule contains the remaining routines.

Appendix A contains a detailed description of each service call.

3. System Call Submodule

MTS was designed to allow each user a certain amount of flexibility in the virtual environment provided by the system. In order to implement this feature it was necessary to provide additional service routines to handle this higher level resource management. These additional routines, called system calls, are utilized by the MCP to satisfy terminal requests for changes in a user's environment. It was decided that these same services should also be provided to user programs. This led to the inclusion of system calls in the service module.

The system calls affect changes in a user's virtual environment by acting directly on the status variables contained in the SSB. Normally, two or more parameters are required to specify exactly the action desired. Each parameter is examined closely by one of several validation procedures contained in the system call submodule. Every effort is made to ensure that the request is compatible with the current system state before any changes are made in the SSB.

E. MTS COMMAND PROCESSOR MODULE

The MTS command processor (MCP) was designed to be a completely independent module of MTS (see IV.E.1). The decision was made to implement MCP utilizing a high-level language, namely PL/M. This approach had the following advantages:

(1) Use of the PL/M structured constructs, debugging and self-documenting facilities to assist in the rapid development, checkout and integration of the command processor.

(2) Assistance in simplifying future maintenance and modification tasks associated with the command processor.

(3) Provided a PL/M program which illustrates and tests the system and service call interface requirements of a program with MTS.

Since the MCP interface did not require direct access to the machine resources, the inefficiency of code generation resulting from the use of a high-level language was outweighed by the advantages it provided.

1. Data Structures

The data structures required for a command processor are fairly common, depending on the number and complexity of commands available to the user. Those used by MCP include: a 64 byte command buffer; a command buffer pointer to the next character; length of the input command sequence; a byte vector to hold and transfer the command parameters to the

service module.

The MTS/MCP interface is through the internal interface port located at 1F00H. The data structures required to generate the function call interface are identical to those used by CP/M [4]. An example of the calling procedures can be found in section F.4 of Appendix A. MCP is also required to save the MTS system stack pointer prior to processing the command. To maintain its independence, MCP utilizes its own stack for internal processing. It then restores the system stack pointer prior to exit. This ensures a proper return to the location in the MTS monitor where MCP was initially called.

2. Command Processing

The command processor is called by the MTS monitor to process a system command entered by the terminal user. After saving the system stack pointer and setting up its own stack, the data structures are initialized. The system command is then read into the command buffer using a sequence of MTS service calls. If the command buffer is empty, no further processing is accomplished, MCP returns to the monitor. Assuming there is a command to process, the first non-blank character of the command is read. All other characters in the command name are ignored by MCP since the first character of each command is unique. If the first character is not one of the valid system commands, an INVALID CMD message is sent to the service module for display on the appropriate terminal's status line. If it is

a valid command character, processing of the command sequence continues.

Each command is processed by its own subroutine. Validation of the parameter sequence and syntax is accomplished for each input command (see D.3, Appendix A). An INVALID CMD error message is displayed for any detected error in syntax. The following conversions of input parameter values are performed:

(1) drive letter - range of values is A-H; the ASCII letter character is converted to a binary number between 0 and 7 (e.g. A=0, B=1, etc.).

(2) disk number - since the range of values is from 0-31, no more than two characters may be entered for this parameter; the input ASCII numeric characters are converted to the equivalent binary value.

(3) memory size - range of values is 0-48, thus up to two ASCII numeric characters are converted to the equivalent binary value.

As the parameters are processed, each is saved in the parameter vector. After the command sequence has been interpreted, MCP calls the MTS service module to continue processing the command request. This call includes the appropriate system command number and the parameter or list of parameters required by the syntax of the command. The MTS service module will always return a completion code to the calling routine. MCP is implemented to immediately return this completion code to the service module in the

form of a DISPLAY MSG system call. This results in the appropriate message being displayed to the user upon completion of command processing (see E.3, Appendix A). MCP then restores the system stack and returns control to the MTS monitor.

F. TERMINAL INTERFACE MODULE

The terminal interface module was designed and implemented to provide all functions and facilities required to interface the four Sycor 440 display terminals with the remainder of MTS. It isolates all the terminal functions and data structures into one independent module. There are no external routines accessed and only two MTS global data structures accessed by the terminal module. This provides a high degree of module independence and minimizes the external linkage requirements.

The terminal interface module provides three general services for the Sycor 440/MTS environment:

(1) terminal primitive functions - These are functions which are commonly provided by the terminal hardware of intelligent text display devices. However, the Sycor 440 requires that they be provided by the software. These include such tasks as converting from a keyboard matrix code to ASCII code for input data, blinking and updating the current position indicator (cursor), and scrolling the display.

(2) input key processing - this is processing done under

interrupt control, for each key activation interrupt received. It includes checking for capitalization of alphabetic character, checking for any special MTS command keys and input character echoing.

(3) system interface functions - these functions provide the primary interface point between the terminal module and other MTS modules. They provide the processing of requests from MTS routines for terminal I/O or status information.

The terminal interface module was logically and physically divided into five (5) submodules. Each submodule contains the linkage and macro definitions necessary to interface with other terminal submodules and global system data structures as required. The passing of parameters between procedures and submodules within the terminal module is accomplished entirely through the machine registers. The terminal interface submodules are discussed in the following paragraphs.

1. Data Structures

This submodule provides all the internal data structures used by the terminal interface routines. An important point to keep in mind is that all data structures providing display I/O control are in multiples of four, requiring one for each of the four terminals. Each terminal was assigned an identification number from 0-3.

The primary data structures providing display I/O control include the four 576 byte DMA buffers discussed in

section IV.F.2 and the following:

(1) display base - a data vector containing the base address of each of the 512 byte display buffers (does not include the 64 byte status line).

(2) cursor - a pointer specifying the current display address (from 0-511) at which the cursor character is to be displayed.

(3) current line - a pointer specifying the initial display address of the current line in which the user is entering data. This line has not yet been terminated by a line termination key (see section IV.F.2).

(4) next char - a pointer specifying the initial display address containing the next character in the input line (buffer) to be processed. An input line is defined as a string of up to 512 ASCII characters which has been terminated by a line termination key.

(5) end ibuff - end of the input buffer; points to the display address where a line termination key was received.

(6) terminal status - contains the current status of each terminal's input buffer; the possible values are input waiting, MTS cmd ready, or ibuff empty.

Additional data structures include: a matrix code to ASCII conversion table; an upper or lower case key entry mode indicator; status line base address vector; and data vectors containing status line messages.

2. Utility Routines

This submodule contains the basic utility routines which provide common register manipulation and processing required by the remaining terminal submodules. The only data structure needed by this submodule is a swap position address used for swap cursor processing. These utility functions include: comparing display pointers to determine if they are equal; converting a binary number to an ASCII display code; getting and storing display characters; moving bytes of data from a memory source to a memory destination; and swapping the current cursor position character with the swap position character. This is the basic mechanism used to provide the blinking cursor feature.

3. Terminal Interface Primitives

This submodule contains the routines which provide the primitive functions of the terminal module. These procedures require linkages to all the display control data structures and the status base address vector. The primitive functions include: blanking a specified area in the display buffer; retrieving appropriate display buffer and status line addresses; returning the current terminal status; sending beep and click alerts to terminals; scrolling a specified terminal's display; and updating the current cursor position.

4. Key Processing Routines

This submodule provides one of the basic functions of the terminal module, that of processing each key character entered at a terminal. It requires linkage to most of the terminal data structures and to one global system data structure, the task status contained in the SSB. A bit is set in the task status byte when an MTS system command is entered at a terminal. This submodule is called by the terminal interrupt handler to process the input key code. The following is a summary of the processing accomplished: converting the input matrix code to the appropriate ASCII code; checking for and converting lower to upper case letters if required; checking for any key commands (including all line editing); and if not a key command, the input character is echoed back to the terminal display. A key command is one of the following:

(1) line termination key - either an ERROR RESET or CR key. Checks to see if input can be accepted. If input is already waiting to be processed the termination key is not accepted and a terminal beep alert is sent to the terminal. Otherwise the current line is established as the new input buffer. The appropriate terminal status is set and if it is an MTS command, the MCP bit in the task status byte is set. This ensures that the command processor is called by the monitor to process this command.

(2) line editing and cursor control keys - provides the processing for character delete, line delete, clear

screen, and move cursor left or right. Terminal beep alerts are generated if the function can not be performed.

(3) entry mode key - sets or clears the current alphabetic key entry to the appropriate upper or lower case mode.

5. System Interface Functions

This submodule contains the routines which provide the terminal module interface with the rest of the MTS modules. It provides reading and writing of characters from or to the terminal display buffers; terminal status information (e.g. whether or not there is input waiting); display of status and MTS messages on the terminal status line; control of the periodic blinking of the display cursor character; and clearing the status line when requested. This submodule links to the terminal data structures and to a global system variable which always specifies the terminal number associated with the function request.

VI. CONCLUSIONS AND RECOMMENDATIONS

The primary goal of this thesis project was to determine the most feasible method of integrating the Sycor 440 system into the facilities of the NPS microcomputer laboratory. The aim was to make available the resources of the Sycor 440 in support of microcomputer systems development. The main objective was to provide an environment for the sharing of these resources among multiple users. The Microcomputer Timeshared System was developed to accomplish this resource sharing. The sharing of the 8080 microprocessor through multiprogramming rather than multiprocessing was implemented only because of the hardware configuration of the Sycor 440. The relatively low cost of microprocessors would normally lead to the implementation of a microprocessor-based timeshared system with dedicated CPUs for each user. Thus, the emphasis was placed on the sharing requirements of the remaining system resources. The sharing of the Sycor 440 memory was implemented through the use of swapping, providing up to 48K of RAM for user programs. Sharing of the mini-disk auxiliary storage was provided through the virtual floppy disk concept.

During the research and development of MTS, several areas were identified as potential candidates for enhancement. These areas include: enhancements to the prototype MTS system, enhancements to the Sycor 440

hardware, and ML80 enhancements.

The baseline timeshared system as implemented provides sharing of the Sycor 440 memory and mini-disk resources amongst multiple terminal users. An obvious extension to this basic implementation is the sharing of the remaining dedicated I/O devices (e.g. printer, cassette, etc.) provided by the Sycor 440. These facilities were planned for during the design phase of MTS and interface points provided throughout the implemented baseline system to ensure easy inclusion at some later date. Additional information on suggested design and implementation approaches for these features are contained in Ref. 1.

From the Sycor 440/MTS interface point of view, the main enhancement to the current hardware configuration would be the inclusion of the fast, multi-sector mini-disk controller presently under development at Sycor. To assist in the further integration of the Sycor 440 into the current microcomputer development facilities, the addition of a floppy disk drive would be helpful. It is recommended, however, that the floppy disk interface be an addition to, not a replacement for, the current cassette interface. The cassette driver is necessary in order to maintain compatibility between the Sycor 440 and the 340 debugger. Additional hardware enhancements which would assist in the implemention of a timeshared microcomputer system include a bounds register for memory protection and a mode register with interrupt logic to provide a dual-state machine.

A secondary objective of this thesis project was to

evaluate the performance of ML80 in the development of a reasonably large system. In general, the use of ML80 provided many advantages over assembly language programming without incurring the penalty of inefficient code generation. The algebraic notation provided by the language proved especially convenient in working at the register level. The improved readability of the source code was of great assistance during the development of MTS. The other features of ML80 discussed in section IV.G proved, for the most part, to be very satisfactory. However, as described in chapter V, the relatively small amount of memory available on the system which hosted the compiler caused numerous problems. To provide a completely adequate environment for the development of larger systems, a necessary improvement would be the modification of ML80 to run on a system with more memory available for tables and stacks. An obvious method of implementing this improvement is to enlarge ML80's fixed size stacks and run the compiler under the version of CP/M modified for the Sycor 440/MTS environment. Additional suggestions for enhancements to ML80 include: adding a macro library capability for ease of implementing common macros, modifying EXTERNAL and COMMON declarations to provide linkage to individual identifiers rather than entire modules as is now done, and outputting compiler error messages and symbol table listings to a print file for easier debugging.

APPENDIX A MTS USER'S MANUAL

The purpose of this document is to provide the user with the information necessary to utilize the Microcomputer Timeshared System (MTS). The contents include information on setting up the Sycor 440 System for use with MTS, loading and initializing MTS, and interfacing with the MTS operating system. Sections A and B provide a general description of MTS design concepts and the Sycor 440 System. Section C provides the detailed information necessary to interface the Sycor 440 System and MTS. Section D contains information on the terminal design, key functions, and system commands which enable the terminal user to communicate with MTS. Section E describes the MTS status line display and defines the various messages displayed by MTS. Section F details the services provided a user program by MTS, and the limitations on a user program running in the MTS environment. For more detailed information on operating procedures for the Sycor 440 System consult section G, Ref. 2. The complete MTS design specification and implementation information is contained in Ref. 1.

TABLE OF CONTENTS

A. MTS CONCEPTS AND DEFINITIONS

The acquisition of the Sycor 440 Clustered Terminal Processing System provided an opportunity for development of a shared environment for microcomputer research and development. In response, the Microcomputer Timeshared System (MTS) was designed and built to provide the basic machine interface and system management functions necessary for a shared environment.

The purpose of MTS is to provide an interface between the bare Sycor 440 machine and up to four user tasks executing concurrently. The MTS environment, as viewed by the user, provides all the microprocessor facilities required for microcomputer research and development. From a system point of view, MTS manages the available hardware to ensure that the hardware resources are equitably and efficiently allocated to competing user programs. MTS is designed to interface with a version of CP/M modified to run on the Sycor 440. This enables all systems and programs designed to run with the CP/M operating system to run on the Sycor 440 with minor modifications (such as a change in load address). This includes all the development facilities available with CP/M, such as the context editor, dynamic debugger, assembler, etc. Reference 1 contains a list of references for CP/M and its facilities.

B. SYCOR 440 HARDWARE DESCRIPTION

The Sycor 440 Clustered Terminal Processing System at NPS is composed of a control unit containing a cassette tape drive, four display terminals, a Centronix matrix printer, and a Sycor Model 340 Communications Terminal.

The control unit is the heart of the 440 system. Contained within a waist-high cabinet are random and control logic including two 8080 chips, 64K of random access memory (RAM), interfaces for all peripheral devices, a five megabyte fixed disk, as well as the aforementioned cassette tape drive.

Located together on the front of the control unit are an ON/OFF/RESET keylock and system status panel. Turning the keylock to the RESET position activates a diagnostic bootstrap program located in read-only memory (ROM). This bootstrap program performs several diagnostic tests on the CPU, memory, and system load device (cassette or mini-disk) and then initiates system loading. The status of the diagnostic tests is indicated by a series of red lights on the system status panel. These lights are turned off in sequence as each phase of the test is successfully completed. When all red lights have been turned off, three green lights on the panel will remain lit to indicate that all power supplies are functioning normally. There is also one additional red light at the bottom of the system status panel which only comes on if the temperature inside the control unit cabinet exceeds normal operating limits.

One of the two 8080 chips located in the 440 control unit serves as the system CPU. The 8080 instruction set consists of the 78 data transfer, arithmetic, logical, branch, stack, I/O, and machine control instructions described in Ref. 3. The Sycor 440 provides a comprehensive set of prioritized interrupts including a timer, peripheral device, and auxiliary storage device interrupts. Passing control information and data between the 8080 CPU and peripheral devices is accomplished by utilizing the I/O ports (called latches in Sycor literature) provided on the 8080 chip.

The second 8080 chip found in the control unit acts as a controller for the mini-disk. The mini-disk is a single platter, movable head, fixed disk blocked into 512 byte sectors. There are 800 tracks on the disk with 13 sectors per track. Data transfer between RAM and the mini-disk is via direct memory access (DMA). The mini-disk controller communicates with the host 8080 CPU through a 13 byte disk control block (DCB) located at a fixed location in memory.

Peripherals supported by the Sycor 440 system include bisynchronous and asynchronous communication devices, up to eight display terminals, serial and line printers, and card readers. The NPS configuration has four display terminals consisting of a typewriter-like keyboard and CRT display device. Each terminal displays a DMA image of a 576 byte terminal buffer located in RAM. Keyboard input is accomplished by software translation of a keyboard matrix code into the corresponding ASCII character code. For

hardcopy output the NPS 440 includes a Centronix serial matrix printer.

Several different auxiliary storage devices may be attached to the Sycor 440 in addition to the mini-disk. These include magnetic tape drives, cassette tape drives, and floppy disk drives. The NPS configuration includes a cassette tape drive located in the control unit. This drive provides compatibility between the Sycor 440 system and the Model 340 debugger.

The Model 340 Communications Terminal is a complete system in its own right which is marketed by Sycor for remote job entry (RJE) applications [4]. When utilized as a hardware debugger, the 340 is augmented with 4K of RAM and a backplane coupling to a special wire-wrapped interface board in the 440 control unit. The 340 debugger is provided with a software package which includes provisions for loading and dumping hex format program files between cassette tape and 440 RAM, examination and modification of individual locations in 440 memory, inserting breakpoints and traps in programs executing on the 440, and single-stepping through a program executing one instruction at a time [6].

There are several hardware characteristics of the Sycor 440 system which strongly influenced the implementation of MTS. The most important of these are:

(1) 8080 CPU architecture

(2) terminal design

(3) mini-disk interface

(4) single-state CPU

(5) lack of memory protection

The impact which each characteristic had on the design and implementation of MTS is covered in Ref. 1. For a more detailed discussion of Sycor 440 hardware characteristics see Ref. 2.

C. SYCOR 440/MTS INTERFACE

The Microcomputer Timeshared System was designed and built for use on the Sycor 440 Clustered Terminal Processing System. For this reason MTS depends heavily on specific features of the Sycor implementation of an 8080 based microprocessor. This dependence includes reliance on Sycor supplied software as well as the 440 hardware, but becomes most apparent to the user in the two areas of loading the system and maintaining system files.

1. Loading the System

The MTS object module resides on the mini-disk in a relocatable format acceptable to the Sycor System Loader. The System Loader is called in to memory by setting the internal system definition switch to 3 and turning the ON/OFF/RESET keylock on the control unit to the RESET position. After MTS is loaded execution begins with the initial program load (IPL) module. The query RECOVERY? (Y/N) is displayed at terminal 0. The operator should enter Y if recovery is desired, otherwise N. In the event that the IPL operation is halted due to a file access error (file non-existent or cannot be read) the message IPL ABORTED followed by a system file name will appear at terminal 0. After correcting the problem the operator may reload in the normal manner. When the IPL ABORTED message is accompanied by the HARDWARE ERROR terminal alert it indicates that an abnormal completion code was returned by the mini-disk controller after a read operation. Further investigation

using the Sycor utility programs FIXNAR or ..ZAP may be required to identify the problem [2,5].

To summarize, loading MTS involves the following steps:

(1) set internal system definition switch to 3

(2) turn ON/OFF/RESET keylock to RESET

(3) wait two minutes while mini-disk reaches operating speed and all red lights on control unit status panel go out

(4) respond to RECOVERY? query with N to initialize a new system or Y to recover from the last operating session.

2. Recovery File - .MTSRCVR

MTS supports limited recovery after a user task causes a system crash. The recovery feature is implemented by copying the contents of the system state block (SSB) after each swap to a mini-disk file known as the recovery file. Since the SSB defines the state of the system at any instant, recovery may be accomplished by reloading the SSB from the recovery file, deleting the task causing the crash, and proceeding with normal execution. These actions are performed by the MTS IPL module when the answer to the RECOVERY? query is Y.

Whenever MTS is running, a single-sector file named .MTSRCVR must be listed in the mini-disk directory. In the event this file is deleted, it may be recreated under the Sycor operating system by using the command

```
CREATE .MTSRCVR N=1
```

The contents of the recovery file at the completion of an operating session are only meaningful if recovery will be requested when MTS is next loaded. Therefore, under normal circumstances this file is not needed when MTS is not running.

3. Swap Files - .MTSSWPx

One of the most fundamental requirements on any timesharing system is maintaining independence of user tasks executing concurrently. MTS satisfies this requirement by maintaining physical as well as logical separation of all user tasks in the system. Associated with each of the four terminal tasks is a mini-disk file used to store a core-image of the task when it is waiting for the CPU or blocked pending some I/O operation. At any given instant a task may reside on the disk in its swap file or in memory, but at no time can two or more tasks reside in memory simultaneously.

A task's swap image consists of 17 bytes reserved by MTS for environment and virtual device control data followed by up to 48,896 bytes of user task memory image. Thus, the maximum allowable swap image is approximately 48K bytes. There is no minimum value for the size of a swap image. The swap image size or, equivalently, the amount of memory space available to the user is variable from 0 to 48K. In fact, the user is encouraged to use the smallest swap image which satisfies his requirements as smaller swap images tend to improve system responsiveness.

A 48K swap image will fill 96 sectors on the mini-disk. Therefore each of the four swap files should normally be 96 sectors long. In the event that mini-disk space is limited, or that user tasks do not require a 48K swap image, MTS will automatically adjust to any file size greater than 16K (32 sectors). Sixteen kilobytes was selected as the minimum and default system size since it provides a reasonable amount of memory for running the CP/M operating system. The IPL module performs a size test on each swap file to ensure that it is at least 32 sectors long. MTS cannot be loaded if any swap file is smaller than 32 sectors.

If it becomes necessary to change the size of any or all swap files, the file(s) must first be deleted from the mini-disk directory. This is accomplished under the Sycor operating system using the command

DELETE <filename>

where <filename> may be .MTSSWP0, .MTSSWP1, .MTSSWP2, or .MTSSWP3. The number in each case indicates the terminal with which the file is associated. After the file has been deleted, it may be recreated by using the command

CREATE <filename> N=96

for each file which has been deleted. If swap files smaller than 48K are desired, the value of N in the CREATE command string should be two times the required memory space in kilobytes, but no less than 32.

The contents of the swap files upon completion of an MTS operating session are only meaningful if recovery will

be requested when MTS is next loaded. Under normal circumstances these four files are not required when MTS is not running.

4. Configuration File - .MTSCNFG

As explained in section A.2.c, MTS identifies virtual floppy disks by a logical disk number ranging from 0 to 31. Since each virtual floppy disk actually resides in a mini-disk file created under the Sycor operating system, there must be some mechanism for mapping a logical disk number into a file name contained in the mini-disk directory. This function is performed by the configuration file .MTSCNFG.

The configuration file is made up of 32 entries of thirteen bytes each contained on a single mini-disk sector. Each entry has the format

where FILENAME is the 0 to 8 byte name of the virtual floppy disk file as it appears in the mini-disk directory; KEY is a 0 to 4 byte protection key; PA is the protection attribute of the virtual disk, i.e. 'P' for read/write protection, 'R' for write protection only (restricted access), and blank for no protection. The logical disk number for each entry is simply the position of that entry within the file. For example, the first entry is assigned logical disk number 0, the last entry 31, and the entry which is preceded by 17

other entries becomes number 17.

The configuration file is read by MTS during the initialization process performed by the IPL module. The filename is extracted from each entry and input to a routine which searches the mini-disk directory. When a match occurs the mini-disk address for the file is read from the directory and entered in a virtual floppy disk map table. If no match occurs for a given configuration file entry, the corresponding logical disk number is marked not available. Any subsequent attempt to access that virtual disk will result in the terminal alert DISK NOT AVAIL (E.3).

The configuration file may only be modified when running under the Sycor operating system. Sycor provides a data entry free form mode which allows the terminal operator to examine and modify the contents of the file [5]. Extreme care must be exercised when updating .MTSCNFG to align each entry properly in the file. MTS assumes the file will be in the proper format when read, and makes no attempt to validate individual entries.

Since the information contained in the configuration file is of a permanent nature and can only be recreated with great difficulty, the file .MTSCNFG should never be deleted from the mini-disk file directory. In the event the file is deleted erroneously, a new file may be created under the Sycor operating system using the RESTORE command and a backup cassette labeled ".MTSCNFG DUMP". The command is entered as

```
RUN RESTORE 2=/CSST
```

with .MTSCNFG DUMP mounted in the cassette drive. This command will build a new file directory entry for .MTSCNFG and establish a basic configuration file with 32 entries of the form .DISKx, where x ranges from 0 to 31. This basic file may then be edited as described above to reflect current file names and protection attributes.

5. Virtual Floppy Disk Files

Each virtual floppy disk resides on a block of logically contiguous mini-disk sectors. This block must be allocated using the facilities provided by the Sycor operating system, specifically the command

CREATE <filename> N=<file size>

where <filename> is a 1 to 8 character name to be entered in the mini-disk directory and configuration file, and <file size> is the length of the file in 512 byte mini-disk sectors. For the standard 256K byte floppy disk <file size> equals 512, i.e. (256 * 1024)/512=512.

Where a physical floppy disk has a fixed capacity of 256K bytes, an MTS virtual disk may have any convenient size. MTS assumes that the disk image is made up of contiguous 128 byte floppy disk sectors starting with track 0 sector 1, proceeding through the 26 sectors of track 0 to track 1 sector 1, and so on until the virtual disk file is full. If the virtual disk file size is less than 512 mini-disk sectors, less than 77 floppy disk tracks will be addressable. Conversely, if the file size exceeds 512, then more than 77 tracks will be addressable, up to a maximum of

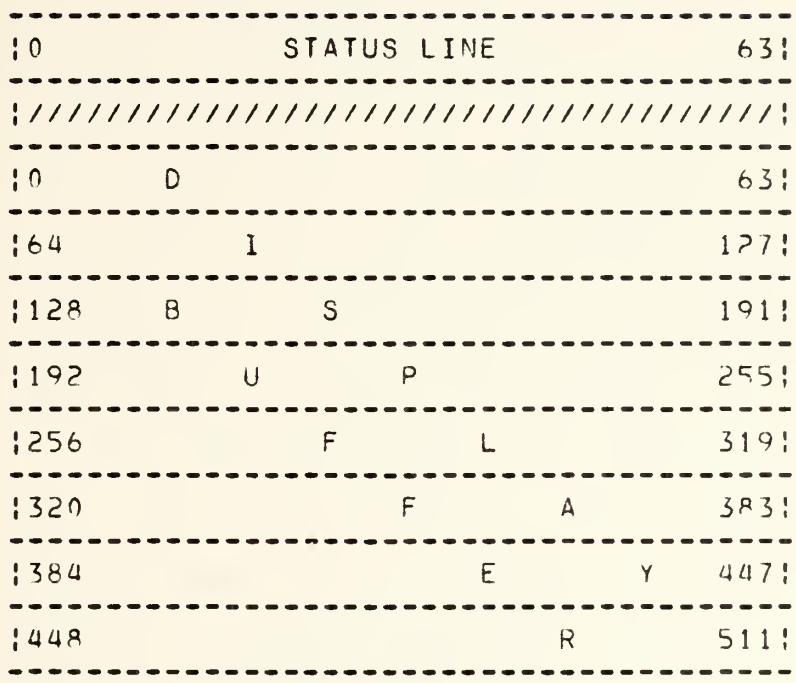
256 tracks. In either case MTS automatically adjusts the upper bound based on the file size as shown in the mini-disk directory.

It is important to note that MTS only recognizes virtual floppy disk files which are entered in the configuration file. The logical disk number associated with a given virtual floppy disk file is determined by that files position in .MTSCNFG. When the file is initially entered in the configuration file a protection key and protection attribute should also be entered, if desired.

D. MTS/USER TERMINAL INTERFACE

1. Terminal Interface Design

The general format of each terminal display is as follows:



The numbers are decimal and specify character positions within the status line and display buffer.

a. Status Line

The terminal status line is used by MTS to display three types of status information:

(1) The current virtual drive and floppy disk assignments for that terminal.

(2) The size of the user's swap image, i.e. the amount of memory space currently available.

(3) Error message alerts produced by MTS system commands, or resulting from user program calls on the DISPLAY

MSG service routine (see F.2.).

The status line display format and contents are discussed in detail in section E.

b. Display Buffer

The display buffer can hold up to a maximum of 512 characters. The display buffer also acts as an input buffer, holding the input data until the user's program requests it. Due to the unavoidable delays in user program response caused by swapping and aggravated by the relatively low data transfer rate of the mini-disk, the MTS terminal interface provides character echoing and simple line editing features. This ensures reasonable response times to key activation by the user. Thus, input data can not be considered available to the user's program until an input line termination character has been received by MTS. To establish an input buffer for a program, the user enters the data and terminates the line by hitting the NEWLINE or ENTER keys on the keyboard. This establishes that line as an input buffer available for processing by the user's program. Note that the key combinations 'I/O CTL M' or 'SHIFT CR' (on the number pad) will also result in the termination of an input line. Either of these keys, as well as NEWLINE and ENTER, may be used for line termination.

Once an input buffer has been established the user may continue to input data on the next line. The user may use any of the line editing or other cursor control features on this new line of input data. However, this new

line may not be terminated until the user's program has processed the previous input buffer (see terminal alerts below).

Each character output from the user's program is displayed at the current cursor position. Each output results in all input buffer pointers being reset to the character position at the end of the output data. Thus, new I/O will start this point. This implies that if the user had been in the middle of entering data when the output occurred, it must be reentered.

c. Terminal Alerts

The MTS terminal interface provides the user with either a visual or audio response to each key depression. Normal visual response is provided by display of the entered character and/or movement of the display cursor. The display cursor is a blinking underscore character which marks the current position on the screen. Data is always entered and displayed at the current cursor position.

The audio responses consist of either a beep or click at the terminal. A terminal beep alert will be generated for any of the following conditions:

(1) An input buffer is waiting to be processed by the user's program and the terminal user attempts to terminate a new input line.

(2) An attempt is made to move the cursor back past the start of the current line. For example, attempting to

delete the previous line or character after the line has been entered by a termination key will result in a beep.

The terminal click alert is associated with the display scrolling feature. Since the display buffer also acts as an input buffer, scrolling the display when the 512 byte display buffer is full could destroy input data which has not yet been processed. For example, the user could be entering a 512 character string. Upon termination of that input line, MTS will lock out scrolling until the user's program has processed the first 64 characters. This ensures that the input data is not destroyed by the scrolling operation. This scrolling lockout is indicated to the user by a terminal click alert.

2. Terminal Key Functions

The terminal keys fall into five basic functional groups: keys for entry of normal character strings; keys which affect the interpretation of the input key character; input line termination keys; line editing and cursor control keys; and number pad keys. These keys and their functions are described in the following subsecions. Within the function descriptions, "current position" refers to the current cursor position on the screen. Any reference to the display of a character in the current position, also implies that the current position is incremented by one.

a. Character String Keys

KEY/SWITCH	FUNCTION
0-9	Displays the input numeric character at the current position on the screen.
Special Characters	Displays the input special character at the current cursor position on the screen.
A-Z	Displays the input alphabetic characters at the current position on the screen. Alphabetic characters are displayed in upper or lower case depending on the key mode (see SHIFT and FS C under Entry Mode Keys).
Tab/Skip	Displays a (horizonal) tab symbol at the current position on the screen.

b. Entry Mode Keys

KEY/SWITCH	FUNCTION
FUNCTION SELECT(FS)	Defines the interpretation of keys for special functions as

defined in this section.

I/O CONTROL (I/O CTL)	Defines the interpretation of keys for special functions as defined in this section. Also used in conjunction with the alphabetic keys to generate appropriate ASCII control codes.
SHIFT	Defines the interpretation of keys used for two (2) characters. Also used to capitalize alphabetic characters when the key entry mode is lower case (see below).
FS C	Sets or clears the alphabetic key entry mode to upper or lower case. Functions as a shift key lock.

c. Line Termination Keys

KEY/SWITCH	FUNCTION
NEW LINE; ENTER; I/O CTL M; SHIFT CR;	Terminates the current line and establishes the just completed input line as an input buffer available for processing by the user's program. The cursor is

	displayed at the left most position of the next line.
ERROR RESET (MTS CMD)	Specifies that the input line which it terminates is to be processed by MTS as a system command.

d. Line Editing and Cursor Control Keys

KEY/SWITCH	FUNCTION
NEXT FMAT (Line Delete)	Deletes all characters from the current position back to the start of the current line.
BACK SPACE (Char Delete)	Deletes the previously entered character.
FS $ or I/OCTL $ (Clear Screen)	Clears the display buffer (not the status line) and leaves the current position at the upper left position of the display buffer.
<--- (Cursor Left)	Moves the current position one to the left. Does not delete previous entry, but allows reentry.
--->(Cursor Right)	Moves the current position one

to the right. Does not delete previous entry, but allows reentry.

e. Number Pad Keys

The number pad keys consist of 10 numeric digits and 8 ASCII control characters located on the right side of the keyboard. The digits function in the same manner as the other numeric digits on the keyboard. The ASCII control characters are displayed when the SHIFT key is depressed in conjunction with the appropriate key. The only control character which affects the display is SHIFT CR (see Line Termination Keys).

3. MTS System Commands

System commands are a set of commands which provide the user with a means of communicating with MTS from the terminal. These commands allow the user to login to MTS; quit MTS; attach virtual floppy disks; protect, restrict, and unprotect virtual floppy disks; and specify the virtual memory size to be used.

a. General Characteristics

A MTS command sequence may be entered anytime after the initialization or reinitialization of MTS. The user enters the desired command sequence, followed by the ERROR RESET key. This signals the operating system that there is an MTS command to be processed. Any errors detected in the command sequence will result in an error

alert message displayed in the MTS message field on the status line. Section E describes the MTS status line display and provides a summary of the error messages.

b. Syntax Rules

The following rules should be used to interpret the syntax for each system command given in section D.3.e.

(1) The command may be entered in upper or lower case. MTS converts the commands to upper case for processing.

(2) Each entry in the command sequence must be separated by one or more spaces.

(3) The entire command name may be used to specify the command. However only the first letter of the command is required, as indicated by the underscore in the syntax. MTS validates only the first letter of the command name.

(4) Parameters are shown in lower case and enclosed by inequality signs (< >). Each parameter name is a variable which must be replaced by the appropriate character string or decimal number entered by the user.

(5) Parameters may be required or optional, depending on the command. Optional parameters are specified by enclosing the parameter in brackets ([ ]). If a parameter is designated as optional, it may be omitted from the command sequence (see section D.3.d).

(6) The designation [<disk nr> [/<key>]] indicates that

the entire parameter sequence is optional, and that <disk nr> may appear without /<key>. The converse, however, is not true.

(7) If parameters are entered in a command sequence they must be in the order specified in the syntax. For example, <disk nr> may not be entered before <drive ltr>.

(8) The notation (error reset) at the end of each command string is a reminder to the user that each MTS command sequence must be terminated by the ERROR RESET key.

c. Parameter Definitions

The system commands have four types of parameters:

(1) <drive ltr> - must be one of the alphabetic characters A through H. It specifies one of the eight virtual disk drives available to a terminal user.

(2) <disk nr> - must be a decimal number in the range 0-31. It specifies one of up to 32 virtual floppy disk files on the Sycor mini-disk.

(3) /<key> - a string of not more than four characters, always preceded by the special character '/' which designates the string as a key parameter. All valid ASCII characters are acceptable including blank, slash (/), and other special characters.

(4) <memory size> - must be a decimal value in the range 0 to 48, which specifies the user's swap image size in kilobytes. The default value for a user swap image is

16K.

d. Default Parameter Values

Certain system commands allow the user to omit the <drive ltr> and/or <disk nr> parameters. In these cases, MTS determines the appropriate drive letter and disk number by scanning its allocation tables for the first available virtual drive or virtual disk, as appropriate. If one is found, it is allocated to the requesting user. Otherwise the appropriate error message is displayed.

The <key> parameter is optional only if the disk requested has no protection attributes specified. Thus there is no default <key> value. As previously mentioned, the default <memory size> parameter is 16K.

e. Command Descriptions

The following pages describe in detail each of the system commands:

(1) ATTACH

(2) LOGIN

(3) PROTECT

(4) QUIT

(5) RESTRICT

(6) SIZE

(7) UNPROTECT

Function:

To attach a virtual floppy disk to a virtual disk drive for use by a user at a specific terminal.

Syntax:

ATTACH [<drive ltr>] [<disk nr> [/<key>]] (error reset)
-

Description:

This system command simulates the physical operation of loading virtual disk <disk nr> into virtual drive <drive ltr>. All parameters are optional, section D.3.d describes the default values when optional parameters are omitted.

Error Messages:

INVALID CMD	- Syntax error in command sequence.
DRIVE NOT AVAIL	- Drive letter has not been specified and there is no drive presently available for assignment.
DISK NR ERROR	- Disk number entered is greater than 31.
DISK IN USE	- Disk number specified is presently allocated with read/write access to another user.
DRIVE LTR ERROR	- Drive letter entered is not one of the letters A through H.
DISK NOT AVAIL	- Either disk number has not been specified and there is no disk presently available for assignment; or the specified disk is not available for assignment.
KEY ERROR	- The specified disk requires a key and either a key has not been entered or the entered key did not match.

Examples:

```
ATTACH A 3 /USR1
A      C
attach 5 /vd#1
a  1
```

Function:

Links the terminal user to MTS and provides the initial load of the user's program or operating system (default system is CP/M).

Syntax:

```
LOGIN [<disk nr> [/<key>]] (error reset)
-
```

Description:

This system command notifies MTS that the requesting terminal is now active, and simulates the physical cold-start bootstrap operation of the user's system. The bootstrap load always takes place from virtual drive A. The virtual disk (and associated key, if any) attached to this drive may be specified as a parameter. The default is disk nr 0, which is a read only disk and always contains the CP/M operating system.

Error Messages:

DISK NR ERROR	- Disk number entered is greater than 31.
DISK IN USE	- Disk number specified is presently allocated with read/write access to another user.
DISK NOT AVAIL	- The specified disk is not available for assignment.
HARDWARE ERROR	- Abnormal completion status was returned by the mini-disk controller following a write operation. This may indicate that the last virtual disk written to was not closed properly and data has been lost.
INVALID CMD	- Syntax error in command sequence.

Examples:

```
LOGIN 3 /D1
L  15
login 25 /d25
1
```

Function:

Adds the read/write protection attribute to the specified virtual disk.

Syntax:

PROTECT <disk nr> /<key> (error reset)
-

Description:

This system command provides the user with the means for on-line assignment of a protection <key> to <disk nr>. This protection may also be added off-line using the Sycor 440 system software (see section C.5).

Error Messages:

DISK NR ERROR - Disk number entered is greater than 31.

HARDWARE ERROR - Abnormal completion status was returned by the mini-disk controller following a read or write operation.

INVALID CMD - Syntax error in command sequence.

KEY ERROR - The specified disk is already protected. To change protection keys use UNPROTECT with current key and then PROTECT with new key.

Examples:

PROTECT 1 /VFD5
protect 10 /key1
p 2 /u#20

Function:

Terminates the terminal user's link to MTS.

Syntax:

```
QUIT (error reset)
-
```

Description:

This system command notifies MTS that the requesting terminal is no longer active.

Error Messages:

HARDWARE ERROR - Abnormal completion status was returned by the mini-disk controller following a write operation. This may indicate that the last virtual disk written to was not closed properly and data has been lost.

INVALID CMD - Syntax error in command sequence.

Examples:

```
QUIT
quit
Q
q
```

Function:

Adds the read restriction attribute to the specified virtual disk.

Syntax:

RESTRICT <disk nr> /<key> (error reset)
-

Description:

This system command provides the user with the means for on-line assignment of a "read only" restriction to <disk nr>. This allows the user to specify a previously protected virtual floppy disk as available to other users for read only access.

Error Messages:

DISK NR ERROR	- Disk number entered is greater than 31.
INVALID CMD	- Syntax error in command sequence.
KEY ERROR	- The specified disk requires a key and either a key has not been entered or the entered key did not match.

Examples:

```
RESTRICT 3 /ID 1
R  4  / ID4
restrict 13 /usr3
r 16 /1
```

Function:

Specifies the memory size to be allocated to the terminal user.

Syntax:

SIZE <memory size> (error reset)
-

Description:

This system command sets the size of the user's swap image. The range of values is 0-48K. The default vaue after login is 16K. is entered.

Error Messages:

INVALID CMD	- Syntax error in command sequence.
OUT OF BOUNDS	- Either the size parameter entered does not fall in the range of 0-48; or the Sycor 440 swap file is not large enough to hold this size swap image (see section C.3).

Examples:

```
SIZE 24
S 32
size   48
s    0
```

Function:

To remove a previously entered protection key from the specified virtual floppy disk.

Syntax:

UNPROTECT <disk nr> /<key> (error reset)
-

Description:

This system command provides the user with the means for on-line removal of all protection attributes from <disk nr>. This protection may also be deleted offline using the Sycor 440 system software (see section C.5).

Error Messages:

DISK NR ERROR - Disk number entered is greater than 31.

HARDWARE ERROR - Abnormal completion status was returned by the mini-disk controller following a read or write operation.

INVALID CMD - Syntax error in command sequence.

KEY ERROR - A protection key is required and either no key was entered, or the entered key did not match.

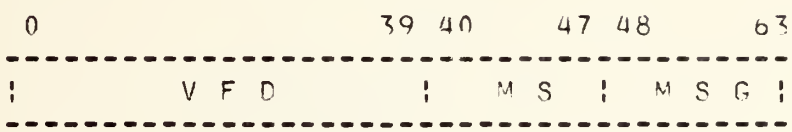
Examples:

```
UNPROTECT 18 /NR 9
U  12  /0964
unprotect 7 /0vfd
u  9  /0?&%
```

E. MTS STATUS LINE MESSAGES

The MTS operating system utilizes the first line of each terminal for system status and error message displays. The status line is 64 characters in length and is divided into three display areas as shown below.

STATUS LINE



where

VFD - Virtual Floppy Disk Status Display

MS - Memory Size Display

MSG - Error Message Display

1. Virtual Floppy Disk Status Display

This display contains information on the virtual drive and disk assignments currently in effect. For each attached virtual floppy disk the following will be displayed:

(1) drive letter

(2) disk number

(3) restriction indicator (r or blank)

For example, if the user has attached disk number 3 to drive A and disk number 25 (which is restricted) to drive C, the status display would appear as follows:

```
 0         10                                        39
-------------------------------------------- - - -
;A=03      C=25r
-------------------------------------------- - - -
```

2. Memory Size Display

The center of the status line display shows the current memory size for that user and the "MTS" header. For example, if the system default memory size were being used, the display would appear as follows:

```
          40      47
- ------------ - - -
      16K MTS
- ------------ - - -
```

3. Error Message Display

All MTS system commands are validated and an error alert generated if any syntax errors are found. The last 16 positions of the status line are reserved for these messages. A valid system command will clear the error message display of any previous error alert. The following is a summary of system error messages.

MESSAGE	MEANING
(Blank Display)	Initial condition; also the status message area is cleared following the processing of a valid system command.
DISK IN USE	Disk number specified is presently allocated with read/write access to another user.

DISK NOT AVAIL	Either the disk number has not been specified and there is no disk presently available for assignment; or specified disk number is not available for assignment.
DISK NR ERROR	Disk number entered is greater than 31.
DRIVE LTR ERROR	Drive number entered is not one of the letters A through H.
DRIVE NOT AVAIL	Drive letter has not been specified and there is no drive presently available for assignment.
HARDWARE ERROR	Abnormal completion status was returned by the mini-disk controller following a read or write operation. May indicate hardware errors on peripheral devices as additional devices are included in the system.
INVALID CMD	Syntax error has been detected in the command sequence.
KEY ERROR	The specified disk requires a key and either a key has not been entered or the entered key did not match.
OUT OF BOUNDS	A numeric parameter has been specified which exceeds the valid bounds for that parameter.

TASK DELETED — When RECOVERY is specified during system initialization, this message is displayed at the terminal which was executing when the system failure occurred. It indicates that this terminal user must reestablish the environment.

F. MTS/USER PROGRAM INTERFACE

1. Program Interface Design

MTS was designed to provide a timeshared, virtual 8080 microprocessor environment for microcomputer systems development. The term virtual is appropriate here because the user actually interfaces with MTS for many services normally provided by hardware in a dedicated CPU environment. A software interface between user programs and the Sycor 440 hardware is necessary in order to allocate the hardware resources equitably and efficiently, while at the same time satisfying the service requirements of several competing user programs.

The MTS/user program interface consists of a set of service routines which may be called by a user program through a single entry point to perform terminal I/O, access virtual floppy disks, or modify the user's virtual environment. The design was heavily influenced by the CP/M operating system which uses a similar scheme for I/O.

The set of service routines may be logically divided into two types of calls on MTS. The first type, system calls, perform the same functions for a user program as system commands provide for the user at a terminal (D.3). The functions deal with modifying the user's current virtual environment by changing memory size, attaching various virtual disks to virtual drives, or even logging on and off the system. Service calls are the second type of call provided by the MTS software interface. Service calls are

used to perform terminal I/O and access virtal floppy disks.

A call to MTS takes the form

<value> = MTS(<fid>,<parm>)

The first argument, <fid>, is a number from 0 to 17 which identifies the function requested. The <parm> argument may be a parameter value, if only a single parameter is required, or the address of a parameter list if more than one parameter is required. In each case MTS returns <value> upon completion of the requested operation. This returned value may be an ASCII character code, an error code, or in several cases have no significance. Both system calls and service calls are formed as described above. The syntax and function of each call are described in the following sections.

2. System Calls

All system commands available to a user at a terminal are also available to a user program through system calls. An additional call is provided which will display an appropriate terminal alert at the user's terminal if entered with an error code. Table 1 summarizes the required arguments and return values of each system call.

TABLE 1

SYSTEM CALL SUMMARY

FID	NAME	PARM	VALUE
0	ATTACH	list	error code
1	DISPLAY MSG	error code	none
2	LOGIN	list	error code
3	PROTECT	list	error code
4	QUIT	none	none
5	RESTRICT	list	error code
6	SIZE	size	error code
7	UNPROTECT	list	error code

a. Arguments

Each system call is identified by a number which MTS associates with a particular service routine. In addition to this function identifier, MTS may require one or several additional parameters to perform the requested service. When more than one parameter is required, MTS must be passed the address of a byte vector containing these parameters. Each system call requires that this vector

conform to some fixed format. In general each byte of the vector will contain some numeric value or an ASCII character code, but situations may arise when an optional parameter is not specified. In such cases the corresponding byte in the parameter vector must be filled with the value FFH.

b. System Call Descriptions

The following pages describe in detail each system call.

Function:

To attach a virtual floppy disk to a virtual disk drive for use by a user at a specific terminal.

Arguments:

FID = 0

PARM = address of parameter vector
byte 0: drive number where A=0, B=1, etc.
byte 1: disk number - 0 to 31
bytes 2-5: protection key - 0 to 4 ASCII characters

Description:

This call simulates the physical operation of loading virtual disk <disk nr> into virtual drive <drive nr>. All parameters are optional. If disk and/or drive nr is not specified MTS searches the disk or drive map table respectively for the first available entry. A protection key is only required if the virtual disk to be attached has been assigned read/write protection. The call returns an error code upon completion.

Error Codes:

1 - Operation successful
2 - Either disk number has not been specified and there is no disk presently available for assignment; or the specified disk is not available for assignment.
3 - Disk number specified is presently allocated with read/write access to another user.
4 - Disk number specified is greater than 31.
5 - The specified disk requires a key and either a key has not been entered or the entered key did not match.
6 - Drive number specified is greater than 7.
10 - Drive number has not been specified and there is no drive presently available for assignment.

Function:

To display a terminal alert in the status message area of the user's terminal.

Arguments:

FID = 1

PARM = error code

Description:

This call takes an error code as input and displays the corresponding predefined terminal alert message on the user's terminal. The DISPLAY MSG system call provides the only way for a user to display messages on the terminal status line. No value is returned by this system call.

Action:

0 - blank
1 - INVALID CMD
2 - DISK NOT AVAIL
3 - DISK IN USE
4 - DISK NR ERROR
5 - KEY ERROR
6 - DRIVE LTR ERROR
7 - OUT OF BOUNDS
8 - HARDWARE ERROR
9 - TASK DELETED
10 - DRIVE NOT AVAIL

Function:

Reinitializes the user's MTS environment and reboots the user's system from drive A.

Arguments:

FID = 2

PARM = address of parameter vector
byte 0: disk number - 0 to 31
bytes 1-4: protection key - 0 to 4 ASCII characters

Description:

This system call creates a reinitialized MTS environment for the user program. Memory size is set to 16K and the specified disk, if any, will be attached to drive A. If no disk number is specified, disk number 0 containing the CP/M system is attached. Drive assignment and memory size will be displayed on the status line of the user's terminal. The user system is then rebooted from drive A.

Error Codes:

0 - Operation successful
2 - The specified disk is not available for assignment.
3 - Disk number specified is currently allocated with read/write access to another user.
4 - Disk number specified is greater than 31.
5 - The specified disk requires a key and either a key has not been entered or the entered key did not match.
8 - Abnormal completion status was returned by the mini-disk controller following a write operation. This may indicate that the last virtual disk written to was not closed properly and data has been lost.

Function:

Logs the user off MTS.

Arguments:

FID = 4

PARM = none required

Description:

This system call notifies MTS that the requesting user program is no longer active. Control will not be returned to the user program. The user must log in through the terminal to regain system facilities.

Error Codes:

0 - Operation successful.

8 - Abnormal completion status was returned by the mini-disk controller following a write operation. This may indicate that the last virtual disk written to was not closed properly and data has been lost. If this error code is returned, the terminal alert HARDWARE ERROR is automatically displayed in the status message area of the user's terminal.

Function:

Set the amount of memory available to the user.

Arguments:

FID = 6

PARM = memory size in kilobytes

Description:

This system call adjusts the size of the user's swap image to the specified value. The value must fall in the range 0 to 48, and also must not be greater than the size of the swap file associated with the user's terminal.

Error Codes:

0 - Operation successful.

7 - Either specified size exceeds 48K, or a value less than 48K exceeds the size of the available swap file.

3. Service Calls

The MTS environment currently provides a virtual CRT terminal as the primary I/O device and virtual floppy disk drives for auxiliary storage. Access to both of these virtual devices is through MTS service calls. A summary of the service calls showing parameters and returned values is given in Table 2.

TABLE 2

SERVICE CALL SUMMARY

FID	NAME	PARM	VALUE
8	TERMINAL STATUS	none	true/false
9	READ TERMINAL	none	character
10	WRITE TERMINAL	character	none
11	WRITE PRINTER	character	none
12	SELECT DRIVE	drive nr	error code
13	SET DMA	dma address	error code
14	SET TRACK	track nr	none
15	SET SECTOR	sector nr	error code
16	READ FLOPPY	none	none
17	WRITE FLOPPY	none	none

a. Virtual Terminals

The MTS virtual terminal simulates the operation of a serial half-duplex console device. Single ASCII characters may be passed from the terminal keyboard to a user program, or from a user program to the terminal for

display. A service call to MTS is required to pass each character. MTS also provides a terminal status service call which allows a user program to test the status of a terminal.

The user should keep in mind that characters are actually being passed between his program and the terminal display buffer (D.1.b). This means that input need not be echoed by the user's program since it already appears on the display. Simple line editing is also provided by MTS on the input data prior to making that data available for processing by the user's program.

The user can directly contribute to improved system response by proper use of the terminal service calls. It is common practice when writing conversational programs to implement a "get character" routine to handle input from the terminal. Normally this routine does little more than repeatedly test the terminal status until it finds input waiting. In the MTS environment a more efficient method of accomplishing the same goal is to immediately read from the terminal without testing for status. If input is waiting, the first character will be passed immediately. More importantly, if there is no input waiting, MTS will block the user's program until a character is entered at the keyboard. The blocked program may be swapped out and the CPU allocated to another user. This method of implementing conversational programs takes advantage of unproductive waiting time in one user program to service additional users.

b. Virtual Floppy Disk Drives

The MTS virtual floppy disk drive provides auxiliary storage for user programs on virtual floppy disks. These hard-sectored disks have 128 bytes per sector, 26 sectors per track, and a maximum number of tracks determined by the size of the file containing the disk image (C.5). Each user has eight drives available for dedicated use.

Drive A is activated when the user logs in and serves as the user system load device. In a process which simulates a cold-start bootstrap load the first four sectors on track 0 are read into the user's memory space at location 4000H. MTS assumes that these sectors contain executable code which will load the remainder of the user's system. Unless another disk is specified in the LOGIN command string, a read-only disk containing the CP/M operating system will be attached when drive A is activated.

The user may activate any or all of the remaining virtual drives by attaching a virtual disk. This is accomplished from the terminal by entering the ATTACH system command or directly from the user's program by a call to MTS. Although no direct method for detaching a virtual disk is provided by MTS, the same effect is produced indirectly by overriding the current drive assignment with a second ATTACH command. When the second floppy disk is attached MTS closes the previously attached disk and releases it for use elsewhere.

Data transfer between a virtual disk and a user program utilizes a 128 byte buffer in the user's program

space called a DMA buffer. The name is derived from the nature of the transfer operation: to the user program it appears that data transfer is by direct memory access.

Before the user program can access a particular virtual disk sector the user must specify a complete sector address and a DMA buffer. A complete sector address consists of drive, track, and sector numbers. Note that MTS will not allow a virtual drive to be selected until a disk has been attached. A DMA buffer is defined by its base address. MTS provides a service call to enter each of these four values. Once a value has been entered it will be used for all subsequent virtual disk accesses until redefined by a second service call.

c. Arguments

Service calls have the same form as other calls to MTS. A numerical function identifier is associated with each call to identify the service desired. The second argument is a single parameter in most cases, although several of the service calls require no second argument.

d. Service Call Descriptions

The following pages describe in detail each individual service call.

Function:

Interrogate the status of the user's terminal.

Arguments:

FID = 8

PARM = none required

Description:

This service call returns a logical value answering the question "Is terminal input waiting?" TERMINAL STATUS should not be used in those situations where no further processing can be accomplished until terminal input is available. In such a case it is more efficient to use the READ TERMINAL service call to allow processing of other user tasks while waiting.

Value:

00H - all terminal input processed

FFH - terminal input waiting

Function:

Read the next available character from the user's terminal.

Arguments:

FID = 9

PARM = none required

Description:

This service call passes the next available ASCII character from the user's terminal display buffer to the user program. The maximum size input line is 512 characters. Each input line is terminated by a carriage return. It is not necessary for user programs to echo input characters since they are already displayed on the user's terminal before becoming available to the user program. Line editing functions are provided by MTS.

Value:

A single ASCII character - the end of each input line is indicated by the return of a carriage return (ASCII = 0DH).

Function:

To write a character to the user's terminal.

Arguments:

FID = 10

PARM = a single ASCII character

Description:

This service call passes the specified character from the user program to the terminal display buffer for display. Carriage return (ASCII = 0DH) returns the cursor to first position of the current line. Line Feed (ASCII = 0AH) moves the cursor down one line. Each output line will normally be terminated by the CR-LF combination.

Value:

None returned.

Function:

Selects the virtual floppy disk drive to be used in subsequent floppy disk accesses.

Arguments:

FID = 12

PARM = drive number where A=1, B=2, etc.

Description:

This service call selects one of the eight virtual floppy disk drives available to each user program for use in subsequent floppy disk accesses. Before a drive can be selected, a virtual disk must be attached.

Error Codes:

0 - Operation Successful.

6 - Drive number specified is greater than 7. Selected drive is changed.

10 - Drive specified is not in use. Indicates that no virtual disk has been attached to the specified drive. Selected drive is unchanged.

Function:

Sets the base address of the 128 byte DMA buffer to be used in subsequent floppy disk accesses.

Arguments:

FID =13

PARM = base address of DMA buffer

Description:

The DMA buffer required to access a virtual floppy disk must be a conticuous block of 128 bytes located in the user's memory space, i.e. with base address greater than 4000H. Specifying a DMA address greater than or equal to FF00H will have unpredictable results, but can normally be expected to cause a system crash and subsequent deletion of the user's task upon recovery.

Error Codes:

0 - Operation successful.

7 - Address specified is less than the base of user's memory space. Current DMA address remains unchanged.

Function:

Sets the floppy disk track number to be used in subsequent virtual floppy disk accesses.

Arguments:

FID = 14

PARM = track number - 0 to 256

Description:

This service call sets the track number to be used in subsequent floppy disk accesses. Values may range from 0 to 256. The value cannot be validated until it is associated with a virtual floppy disk number; therefore, no validation is performed until a read or write operation is requested.

Value:

None returned.

Function:

Sets the floppy disk sector number to be used in subsequent virtual floppy disk accesses.

Arguments:

FID = 15

PARM = sector number - 1 to 26

Description:

This service call sets the sector number to be used in subsequent virtual floppy disk accesses. Since each floppy disk track contains 26 sectors numbered from 1 to 26, this value cannot be less than 1 nor greater than 26.

Error Codes:

0 - Operation successful.

7 - Sector number specified is less than 1 or greater than 26. Current value of sector number remains unchanged.

Function:

Simulates reading a 128 byte sector from a floppy disk.

Arguments:

FID = 16

PARM = none required

Description:

This service call simulates reading from a floppy disk by mapping the current drive, track, and sector numbers into a mini-disk address; reading the mini-disk sector into a buffer; and moving 128 bytes from the mini-disk buffer into the current DMA buffer in the user's memory space. Errors may occur at two points in the process. If the calculated mini-disk address falls outside the bounds of the virtual disk file attached to the specified virtual drive, it indicates an error in the specified track number. Errors may also occur during mini-disk read and write operations. A user program must consider such hardware errors as irrecoverable since MTS provides insufficient information to determine the cause.

Error Codes:

0 - Operation successful.

7 - Calculated mini-disk address out of bounds. Probable error in specified track number.

8 - Abnormal completion status was returned by the mini-disk controller following a read or write operation.

Function:

Simulates writing a 128 byte sector to a floppy disk.

Arguments:

FID = 17

PARM = none required

Description:

This service call simulates writing to a floppy disk by mapping the current drive, track, and sector numbers into a mini-disk address; reading the mini-disk sector into a buffer; and moving 128 bytes of data from the current DMA buffer in the user's memory space into the mini-disk buffer. Errors may occur at two points in the process. If the calculated mini-disk address falls outside the bounds of the virtual disk file attached to the specified virtual drive, it indicates an error in the specified track number. Errors may also occur during mini-disk read and write operations. The user should interpret such hardware errors as indicating a bad sector on the virtual floppy disk and try repeating the operation with a different floppy disk sector and track combination.

Error Codes:

0 - Operation successful.

7 - Calculated mini-disk address out of bounds. Probable error in specified track number.

8 - Abnormal completion status was returned by the mini-disk controller following a read or write operation.

4. Calling Procedure

All calls to MTS, whether system calls or service calls, are made through a single entry point at location 2000H. Each call takes two arguments: the function identifier in register C; and a parameter value or address in register pair DE. In those cases where the second argument is only a single byte the contents of the D register are ignored.

Each call to MTS returns a value in the A register. This value may be an error code, an ASCII character code, or zero. The value zero is returned by those routines whose value has no significance such as WRITE TERMINAL or SET TRACK.

Note that the register assignments for arguments and returned values conform to the PL/M convention for passing parameters. The following examples illustrate the calling procedure for 8080 Assembly Language, ML80, and PL/M in the MTS environment. Each example illustrates the sequence required to read floppy disk sector 22, track 43 on drive 2 into a DMA buffer at address 4100H.

a. 8080 Assembly Language

When writing in 8080 assembly language MTS is accessed by a direct call to the MTS entry point:

```
MTS     EQU     2000H
        .
        .
        .
        MVI     C,12        ;FID = 12
        MVI     E,2         ;DRIVE NR = 2
```

```
        CALL    MTS         ;SELECT DRIVE
        MVI     C,13        ;FID = 13
        LXI     D,4100H     ;DMA ADDRESS = 4100H
        CALL    MTS         ;SET DMA
        MVI     C,14        ;FID = 14
        MVI     E,43        ;TRACK NR = 43
        CALL    MTS         ;SET TRACK
        MVI     C,15        ;FID = 15
        MVI     E,22        ;SECTOR NR = 22
        CALL    MTS         ;SET SECTOR
        MVI     C,16        ;FID = 16
        CALL    MTS         ;READ FLOPPY
        .
        .
        .
        MVI     C,15        ;FID = 15
        MVI     E,23        ;CHANGE SECTOR NR
        CALL    MTS         ;SET SECTOR
        MVI     C,16        ;FID = 16
        CALL    MTS         ;READ AGAIN
```

b. ML80

The readibility of ML80 source programs may be enhanced by defining an M80 macro for each call to MTS used in the program. The following code segment contains several examples.

```
[MACRO MTS '2000H']
[MACRO SELECT$DRIVE DNR '
    C = 12; E = [DNR]; CALL [MTS]']
[MACRO SET$DMA DMA '
    C = 13; DE = [DMA]; CALL [MTS]']
[MACRO SET$TRACK TNR '
    C = 14; E = [TNR]; CALL [MTS]']
[MACRO SET$SECTOR SNR '
    C = 15; E = [SNR]; CALL [MTS]']
[MACRO READ$FLOPPY '
    C = 16; CALL [MTS]']
    .
    .
    .
/* SPECIFY COMPLETE SECTOR ADDRESS AND DMA BUFFER */
[SELECT$DRIVE '2'];
[SET$DMA '4100H'];
[SET$TRACK '43'];
[SET$SECTOR '22'];
[READ$FLOPPY];
    .
```

```
    .
    .
/* INCREMENT SECTOR NR AND READ AGAIN */
[SET$SECTOR '23'];
[READ$FLOPPY];
```

c. PL/M

```
/*************************************************************/
/*        SAMPLE PL/M PROGRAM SEGMENT                        */
/*************************************************************/

4000H:

USER: PROCEDURE;
DECLARE
    LIT           LITERALLY 'LITERALLY',
    MTS           LIT       '2000H',
    SELECT$DRIVE  LIT       '12',
    SET$DMA       LIT       '13',
    SET$TRACK     LIT       '14',
    SET$SECTOR    LIT       '15',
    READ$FLOPPY   LIT       '16',
    DISPLAY$MSG   LIT       '1' ;
/*************************************************************/
/*             MTS INTERFACE PROCEDURES                      */
/*************************************************************/

MTS1: PROCEDURE (FID,PARM);
    /*************************************/
    /* PROVIDES THE MTS INTERFACE FOR    */
    /* FUNCTIONS WHICH DO NOT REQUIRE A  */
    /* RETURN VALUE.                     */
    /* INPUT: FID - MTS FUNCTION ID      */
    /*        PARM - PARAMETER OR ADDRESS*/
    /*               OF PARAMETER LIST.  */
    /*************************************/
    DECLARE FID BYTE, PARM ADDRESS;
    GO TO MTS;
    END MTS1;

MTS2: PROCEDURE (FID,PARM) BYTE;
    /*************************************/
    /* PROVIDES THE MTS INTERFACE FOR    */
    /* FUNCTIONS WHICH REQUIRE A VALUE   */
    /* RETURNED. INPUT PARAMETERS ARE    */
    /* THE SAME AS IN MTS1.              */
    /*************************************/
    DECLARE FID BYTE, PARM ADDRESS;
    GO TO MTS;
    END MTS2;
    .
```

```
     .
     .

/*********************************************************/
/*  SPECIFY COMPLETE SECTOR ADDRESS AND DMA BUFFER  */
/*********************************************************/

CALL MTS1(SELECT$DRIVE, 2);
CALL MTS1(SET$DMA, 4100H);
CALL MTS1(SET$TRACK, 43);
CALL MTS1(SET$SECTOR, 22);
/*********************************************************/
/* READ FLOPPY RETURNS AN ERROR CODE WHICH WILL      */
/* BE RETURNED TO MTS TO BE DISPLAYED ON THE         */
/* TERMINAL STATUS LINE.                             */
/*********************************************************/
CALL MTS1(DISPLAY$MSG, MTS2(READ$FLOPPY,0));
     .
     .
     .
/* INCREMENT SECTOR NR AND READ AGAIN                */

CALL MTS1(SET$SECTOR, 23);
CALL MTS1(DISPLAY$MSG, MTS2(READ$FLOPPY,0));
     .
     .
     .
END USER;
     .
     .
     .
EOF
```

5. Limitations on User Programs

MTS was designed to provide each user with his own virtual 8080 microprocessor. Unfortunately, the architecture of the 8080 CPU is not amenable to self-virtualization. As a consequence several limitations must be imposed on user programs running in the MTS environment. These limitations are:

(1) The user's memory space extends from address 4000H to FEFF, a total of 48,896 bytes. All user code, data, and buffers must be contained within this area of memory.

(2) All user-defined stacks must be four bytes longer than the maximum size required by the user. The four extra bytes are needed if an interrupt occurs while the user's stack is full. Failure to provide this additional space may result in random execution errors which are not reproducible and extremely difficult to diagnose.

(3) User programs should not read or write directly to I/O ports while running under MTS. Terminal and floppy disk access is provided by MTS service calls. Attempts to interface directly with the Sycor 440 peripherals or auxiliary storage devices may interfere with the operation of MTS and damage other users.

G. REFERENCES

1. Brown, K. J. and Bullock, D. R., A Shared Environment for Microcomputer System Development, M.S. Thesis, NPS, Monterey, CA, March 1977.

2. Brown, K. J. and Bullock, D. R., Hardware Characteristics of the Sycor 440 Clustered Terminal Processing System, unpublished research notes, NPS, Monterey, CA, March 1977.

3. Intel Corp., 8080 Assembly Language Programming Manual, Santa Clara, CA, 95051, 1976.

4. Sycor Inc., Model 340/340D Intelligent Communications Terminal Operator's Manual, Nr TD34003-275, Ann Arbor, Michigan, Oct. 1974.

5. Sycor Inc., User's Guide to the 440 System, Nr UG4400-2, Ann Arbor, Michigan, November 1976.

6. Sycor Inc., 8080 Debug User's Guide, Spec 950706, Rev B, Ann Arbor, Michigan, October 1975.

```
/*********************************************************/
/*                  GLOBAL IDENTIFIERS                   */
/*********************************************************/
/*                                                       */
/* THE FOLLOWING DECLARATIONS DEFINE SYSTEM IDENTI-      */
/* FIERS WHOSE SCOPE IS GLOBAL THROUGHOUT MTS. THESE     */
/* IDENTIFIERS MAY BE DIVIDED INTO THREE DISTINCT        */
/* GROUPS. THE FIRST GROUP INCLUDES ANY IDENTIFIER       */
/* CONSIDERED GLOBAL BECAUSE IT IS REFERENCED IN TWO     */
/* OR MORE MODULES OF THE MTS ML80 SOURCE PROGRAM. BY    */
/* INCLUDING THE DECLARATIONS FOR ALL SUCH IDENTIFIERS   */
/* IN A SINGLE MODULE, INTERMODULE LINKAGE IS GREATLY    */
/* SIMPLIFIED, AND THE SOURCE PROGRAMS'S READABILITY     */
/* AND CLARITY ARE IMPROVED.                             */
/*                                                       */
/* THE SECOND GROUP OF IDENTIFIERS INCLUDES THOSE        */
/* VARIABLES WHICH, TAKEN TOGETHER, DEFINE THE STATE     */
/* OF THE SYSTEM, I.E. THE SYSTEM STATE BLOCK. THE       */
/* CONCEPT OF SYSTEM STATE IS IMPORTANT IN MTS BECAUSE   */
/* THE SYCOR 440 HARDWARE ARCHITECTURE PROVIDES NO       */
/* PROTECTION AGAINST INADVERTENT OR MALICIOUS TAMP-     */
/* ERING WITH SYSTEM CODE BY USER PROGRAMS. TO MINI-     */
/* MIZE THE EFFECTS OF SYSTEM CRASHES CAUSED BY SUCH     */
/* TAMPERING MTS PROVIDES A LIMITED RECOVERY CAPABIL-    */
/* ITY. AFTER A TASK'S TIMESLICE EXPIRES, AND JUST       */
/* PRIOR TO INITIATING A NEW TASK, THE MTS MONITOR       */
/* COPIES THE CONTENTS OF THE SYSTEM STATE BLOCK TO A    */
/* FILE NAMED .MTSRCVR. IF A CRASH OCCURS WHILE THE      */
/* NEW TASK IS EXECUTING, RECOVERY MAY BE ACCOMPLISHED   */
/* BY REBOOTING MTS AND READING THE CONTENTS OF          */
/* .MTSRCVR BACK INTO THE SYSTEM STATE BLOCK. THE        */
/* TASK WHICH CAUSED THE CRASH IS THEN DELETED AND       */
/* NORMAL OPERATION CONTINUES.                           */
/*                                                       */
/* THE THIRD AND FINAL GROUP OF IDENTIFIERS INCLUDES     */
/* SYSTEM DATA ASSOCIATED WITH A PARTICULAR USER TASK.   */
/* SINCE THIS DATA IS ONLY USED WHEN ITS ASSOCIATED      */
/* TASK IS ACTIVE, THE SPACE REQUIRED FORMS A SYSTEM     */
/* AREA IN THE TASK'S SWAP IMAGE. THIS DATA IS SWAPPED   */
/* IN AND OUT ALONG WITH THE USER AREA OF THE SWAP       */
/* IMAGE.                                                */
/*                                                       */
/* THE THREE PRIMARY IDENTIFIER GROUPS DESCRIBED         */
/* ABOVE MAY ALSO BE SUBDIVIDED BASED ON USAGE AND       */
/* STORAGE ALLOCATION REQUIREMENTS. THE GROUP AND        */
/* SUBGROUP HEADINGS FOR DECLARATIONS IN THIS MODULE     */
/* ARE AS FOLLOWS:                                       */
/*                                                       */
/*      A.  GENERAL SYSTEM DECLARATIONS                  */
/*      B.  SYSTEM STATE BLOCK DECLARATIONS              */
/*          1.  SYSTEM CONTROL                           */
/*          2.  TASK CONTROL TABLE                       */
/*          3.  DISK MAP TABLE                           */
/*      C.  SYSTEM SWAP AREA DECLARATIONS                */
/*          1.  VIRTUAL DISK CONTROL BLOCK               */
/*          2.  SWAP STACK                               */
/*                                                       */
/* THE ORDER OF ALL DECLARATIONS IN THIS MODULE MUST     */
/* BE MAINTAINED TO PRODUCE A PROPERLY FORMATTED         */
/* OBJECT MODULE. IN THIS REGARD SPECIFICATION OF THE    */
/* INITIAL ATTRIBUTE IN A DECLARATION MUST BE CONSID-    */
/* ERED CAREFULLY SINCE THE ML80 COMPILER ALLOCATES      */
/* DIFFERENT AREAS OF MEMORY FOR INITIALIZED AND         */
/* UNINITIALIZED VARIABLES. SPECIAL PRECAUTIONS ARE      */
/* ALSO NECESSARY FOR LOCAL VARIABLES USED ONLY IN       */
/* SINGLE MODULES. THE ML80 LINK EDITOR IS FORCED TO     */
```

```
/* ALLOCATE SPACE FOR SUCH VARIABLES WITHIN THE        */
/* MODULE'S CODE AREA BY DECLARING EACH SUCH VARIABLE */
/* WITH TYPE DATA. THIS TECHNIQUES IMPOSES A PENALTY  */
/* OF THREE BYTES PER DECLARATION FOR UNNECESSARY     */
/* JUMP INSTRUCTIONS, BUT THE SIMPLIFICATION OF       */
/* INTERMODULE LINKAGES MAKES THE TRADEOFF WORTHWHILE.*/
/*                                                    */
/******************************************************/
/******************************************************/


/******************************************************/
/************* GENERAL SYSTEM DECLARATIONS *************/
/******************************************************/


DECLARE PARM(2) BYTE INITIAL (0,0);
DECLARE DISK BYTE INITIAL (0);
DECLARE DRIVE BYTE INITIAL (0);
DECLARE ERROR BYTE INITIAL (0);
DECLARE LOCK BYTE INITIAL (1);
    /* SYSTEM LOCK -                                  */
    /*     BIT 0: SWAP LOCK - MTS CODE EXECUTING      */
    /*     BIT 1: SPOOL LOCK - SPOOLING UNDER         */
    /*                  INTERRUPT CONTROL             */
    /*     BITS 2-7:  (NOT USED)                      */
DECLARE TASKSTIMER BYTE INITIAL (0FFH);
    /* COUNTER RECORDING HOW MANY TIMER INCREMENTS    */
    /* (50MS) REMAIN IN TIMESLICE                     */
DECLARE SVCSSTACK(20) BYTE INITIAL (0);
    /* SERVICE MODULE STACK */
DECLARE SYSSSTACK(20) BYTE INITIAL (0);
    /* MONITOR MODULE STACK */
DECLARE MDBUF(512) BYTE INITIAL (0);
    /* MINI-DISK BUFFER - CONTAINS ONE SECTOR */
DECLARE MDSAD(2) BYTE INITIAL (0,0);
    /* SECTOR NUMBER  OF DATA CONTAINED IN MDBUF */


/******************************************************/
/********** SYSTEN STATE BLOCK DECLARATIONS ***********/
/******************************************************/


/******************** SYSTEN CONTROL ********************/

DECLARE TASK BYTE INITIAL (0);
    /* TERMINAL NR OF TASK CURRENTLY ALLOCATED    */
    /* THE CPU - RANGE 0-3                        */
DECLARE RECSFILE(2) BYTE INITIAL (0,0);
    /* MINI-DISK SECTOR NUMBER  OF .MTSRCVR */
DECLARE CNFGSFILE(2) BYTE INITIAL (0,0);
    /* MINI-DISK SECTOR NUMBER  OF .MTSCNFG */

/****************** TASK CONTROL TABLE ******************/
/* THE TCT CONTAINS INFORMATION ON THE STATE OF EACH  */
/* TASK AND DATA REQUIRED TO SUPPORT SWAPPING. EACH   */
/* VARIABLE CONTAINS FOUR ENTRIES - ONE FOR EACH OF   */
/* THE FOUR TERMINAL TASKS.                           */
/******************************************************/

DECLARE TCTSSTATUS(4) BYTE INITIAL (0,0,0,0);
    /*     BIT 0: SIMULATE BOOTSTRAP DURING NEXT  */
    /*              TIMESLICE                     */
    /*     BIT 1: CALL MCP DURING NEXT TIMESLICE  */
    /*     BIT 2:  (RESERVED FOR CASSETTE)        */
    /*     BIT 3:  (RESERVED FOR ASYNC)           */
    /*     BIT 4: BLOCKED FOR PRINTER I/O         */
    /*     BIT 5: BLOCKED FOR TERMINAL I/O        */
    /*     BIT 6: CURRENT SWAP IMAGE RESIDES ON   */
    /*              MINI-DISK                     */
    /*     BIT 7: CURRENT SWAP IMAGE IN MEMORY    */
```

```
DECLARE TCT$DM(32) BYTE INITIAL (0C0H,0,0,0,0,0,0,0,
    0C0H,0,0,0,0,0,0,0,0C0H,0,0,0,0,0,0,0,
    0C0H,0,0,0,0,0,0,0);
    /* DRIVE MAP - POINTERS TO VIRTUAL DISK ASSOC- */
    /* IATED WITH EACH VIRTUAL DRIVE. BYTES 0-7    */
    /* CORRESPOND RESPECTIVELY TO DRIVES A-H FOR   */
    /* EACH 8 BYTE ENTRY.                          */
    /*     BITS 0-4: DISK NR - RANGE 0-31          */
    /*     BIT 5:  (NOT USED)                      */
    /*     BIT 6: READ ONLY FLAG                   */
    /*     BIT 7: IN USE FLAG                      */
DECLARE TCT$SIZE(4) BYTE INITIAL (32,32,32,32);
    /* SIZE OF SWAP IMAGE EXPRESSED IN NUMBER */
    /* OF 512 BYTE MINI-DISK SECTORS          */
DECLARE TCT$BOE(8) BYTE INITIAL (0,0,0,0,0,0,0,0);
DECLARE TCT$EOE(8) BYTE INITIAL (0,0,0,0,0,0,0,0);
    /* MINI-DISK SECTOR NUMBER  FOR EACH SWAP FILE */
    /* - INITIALIZED WHEN MTS LOADED               */

/******************** DISK MAP TABLE ********************/
/* THE DMT CONTAINS INFORMATION ON THE STATUS, PRO-    */
/* TECTION, AND LOCATION ON THE MINI-DISK OF ALL VIR-  */
/* TUAL FLOPPY DISKS. EACH VARIABLE CONTAINS 32        */
/* ENTRIES - ONE FOR EACH POTENTIAL DISK NR. THE       */
/* ENTIRE TABLE IS LOADED AND VERIFIED DURING MTS      */
/* INITIALIZATION.                                     */
/*******************************************************/

DECLARE DMT$FLAG(32) BYTE INITIAL (0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
    /*     BIT 0: DISK EXISTS                       */
    /*     BIT 1: IN USE                            */
    /*     BIT 2: PROTECTED - KEY REQUIRED          */
    /*     BIT 3: RESTRICTED TO READ ONLY W/O KEY   */
    /*     BITS 4-7:  (NOT USED)                    */
DECLARE DMT$BOE(64) BYTE INITIAL (0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0);
    /* MINI-DISK SECTOR NUMBER  FOR BEGINNING */
    /* OF EXTENT                              */
DECLARE DMT$EOE(64) BYTE INITIAL (0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0);
    /* MINI-DISK SECTOR NUMBER  FOR END */
    /* OF EXTENT                        */
DECLARE DMT$KEY(128) BYTE INITIAL (20H);
    /* ONE TO FOUR ASCII CHAR PROTECTION KEY */


/*******************************************************/
/*********** SYSTEM SWAP AREA DECLARATIONS **********/
/*******************************************************/


/************* VIRTUAL DISK CONTROL BLOCK *************/
/* EACH USER TASK HAS AVAILABLE 8 VIRTUAL DRIVES WHICH*/
/* MAY BE SELECTED TO ACCESS THE ATTACHED VIRTUAL      */
/* DISK. FOR EACH USER IT IS NECESSARY TO RECORD       */
/* WHICH DRIVE IS CURRENTLY ACTIVE. AND ADDITIONAL     */
/* DATA NEEDED TO MAP A VIRTUAL DISK ACCESS INTO A     */
/* PHYSICAL MINI-DISK ACCESS. ALL THIS INFORMATION IS */
/* MAINTAINED IN THE VDC BLOCK. THE VDC BLOCK ASSOC-   */
/* IATED WITH EACH TASK IS CONTAINED IN THAT TASKS     */
/* SWAP FILE IN A SPECIAL AREA RESERVED FOR MTS SYS-   */
/* TEM USE. THIS MEANS THAT ONLY ONE OF THE FOUR VDC   */
/* BLOCKS MAINTAINED BY THE SYSTEM IS EVER RESIDENT    */
/* IN MEMORY AT ANY ONE TIME.                          */
/*******************************************************/

DECLARE VDC$DRIVE BYTE;
```

```
    /*     BITS 0-2: DRIVE NR FOR DRIVE CURRENTLY   */
    /*               SELECTED                       */
    /*     BITS 3-5:  (NOT USED)                    */
    /*     BIT 6: READ ONLY FLAG                    */
    /*     BIT 7: MODIFICATION FLAG - SET WHEN CON- */
    /*            TENTS OF BUFFER MODIFIED          */
DECLARE VDC$BOE(2) BYTE;
    /* MINI-DISK SECTOR NUMBER  FOR BOE OF VIRTUAL */
    /* DISK CURRENTLY ATTACHED TO SELECTED DRIVE   */
DECLARE VDC$EOE(2) BYTE;
    /* MINI-DISK SECTOR NUMBER  FOR EOE OF VIRTUAL */
    /* DISK CURRENTLY ATTACHED TO SELECTED DRIVE   */
DECLARE VDC$SECTOR BYTE;
    /* VIRTUAL DISK SECTOR  NR FOR SUBSEQUENT */
    /* ACCESSES - RANGE 1-26                  */
DECLARE VDC$TRACK BYTE;
    /* VIRTUAL DISK TRACK NR FOR SUBSEQUENT */
    /* ACCESSES - RANGE 0-76                */
DECLARE VDC$DMA(2) BYTE;
    /* MEMORY ADDRESS OF 128 BYTE DMA BUFFER */
    /* FOR SUBSEQUENT VIRTUAL DISK ACCESSES  */

/********************* SWAP STACK *********************/
/* EACH TIME A TASK IS SWAPPED OUT THE CURRENT OPER-   */
/* ATING ENVIRONMENT, I.E. PSW, BC, DE, HL, AND SP,    */
/* MUST BE SAVED IN A KNOWN AREA SO THAT IT CAN BE     */
/* QUICKLY RESTORED WHEN THE TASK IS SWAPPED BACK IN.  */
/* MTS USES A STACK IN THE SYSTEM AREA OF THE SWAP     */
/* IMAGE TO HOLD THE ENVIRONMENT WHEN A TASK IS        */
/* INACTIVE.                                           */
/*****************************************************/

DECLARE SWAP$STACK(10) BYTE;
    /* AREA IN WHICH USER ENVIRONMENT IS */
    /* SAVED WHEN TASK IS SWAPPED OUT    */

EOF
```

```
/*************************************************************/
/*                    INTERRUPT MODULE                       */
/*************************************************************/
/*                                                           */
/*                                                           */
/* ALL HARDWARE INTERRUPTS ON THE SYCOR 440 SYSTEM           */
/* CAUSE THE EXECUTION OF A RST 1 INSTRUCTION. THIS          */
/* INSTRUCTION BEHAVES LIKE A CALL TO LOCATION 0008H,        */
/* I.E. THE PC VALUE IS STACKED, AND CONTROL TRANS-          */
/* FERRED TO LOCATION 0008H. DUE TO THIS HARDWARE            */
/* CHARACTERISTIC, THE USER MUST ENSURE THAT ANY USER        */
/* DEFINED STACKS ARE AT LEAST FOUR BYTES LARGER THAN        */
/* THE MAXIMUM SIZE REQUIRED BY THE USER'S OWN CODE.         */
/* SINCE ALL PERIPHERAL DEVICES CAUSE EXECUTION OF           */
/* THE SAME INTERRUPT INSTRUCTION, SOME MEANS MUST BE        */
/* AVAILABLE TO DISTINQUISH BETWEEN DEVICES WHENEVER         */
/* AN INTERRUPT OCCURS. THE SYCOR 440 SOLVES THIS            */
/* PROBLEM BY DEFINING AN INTERRUPT LEVEL FOR EACH           */
/* DIFFERENT DEVICE. THERE ARE 17 INTERRUPT LEVELS           */
/* WITH VALUES RANGING FROM 0 TO 16. A HIGHER NUMERIC        */
/* VALUE ALSO IMPLIES A HIGHER PRIORITY FOR THE              */
/* ASSOCIATED DEVICE. WHEN AN INTERRUPT OCCURS THE           */
/* LEVEL IS AVAILABLE ON INPUT LATCH 0. SIMULTANEOUS         */
/* INTERRUPTS WILL BE INPUT SEQUENTIALLY IN PRIORITY,        */
/* I.E. DESCENDING, SEQUENCE BY LEVEL NUMBER. WHEN           */
/* THE LEVEL READS ZERO ALL PENDING INTERRUPTS HAVE          */
/* BEEN PROCESSED. THE INTERRUPT LEVEL ASSIGNMENTS           */
/* WHICH APPLY TO THE CURRENT NPS SYCOR 440 HARDWARE         */
/* CONFIGURATION ARE AS FOLLOWS:                             */
/*                                                           */
/*          LEVEL          DEVICE                            */
/*          -----          ------                            */
/*            16           DEBUGGER                          */
/*            15           POWER FAIL                        */
/*            14           PARITY CONTROL                    */
/*            11           ASYNC COMM                        */
/*            10           TERMINAL GROUP 0                  */
/*             8           TIMER                             */
/*             6           PRINTER 0                         */
/*             2           FLOPPY DISK                       */
/             1           CASSETTE                          */
/*                                                           */
/* THIS MODULE CONTAINS THE CODE USED BY MTS TO PRO-         */
/* CESS INTERRUPTS. THIS CODE CONSISTS OF AN INTER-          */
/* RUPT CONTROLLER PLUS A SET OF INTERRUPT HANDLER           */
/* ROUTINES - ONE ROUTINE FOR EACH DEVICE. THE INTER-        */
/* RUPT CONTROLLER SAVES THE CURRENT ENVIRONMENT,            */
/* IDENTIFIES THE INTERRUPT LEVEL, CALLS THE APPROP-         */
/* RIATE HANDLER ROUTINE, AND THEN RESTORES THE              */
/* ENVIRONMENT BEFORE RETURNING TO THE INTERRUPTED           */
/* PROGRAM. THE HANDLER ROUTINES ARE TAILORED TO THE         */
/* SPECIFIC REQUIREMENTS OF DIFFERENT DEVICES. IN            */
/* ORDER TO UTILIZE THE CODE CONTAINED IN THE INTER-         */
/* RUPT MODULE IT IS NECESSARY FOR THE MTS INITIAL-          */
/* IZATION ROUTINE TO LOAD A JUMP TO THE INTERRUPT           */
/* CONTROLLER IN MEMORY LOCATIONS 0008-000AH.                */
/*                                                           */
/*                                                           */
/*************************************************************/
/*************************************************************/
```

```
/************ INTERRUPT PROCESSING MACROS ************/

[MACRO LEVEL 'IN(0)']
[MACRO DEBUG$LATCH '0FFH']
[MACRO CSST$LATCH '85H']
[MACRO PRINTER$LATCH '8AH']
[MACRO TIMER$LATCH '02H']
[MACRO TERMINAL$LATCH '3EH']
[MACRO MATRIX$LATCH '3FH']
[MACRO INT$PENDING '(A=[LEVEL]A) !ZERO']
[MACRO DEBUG$INT$PENDING '(A=[LEVEL]; A::16) ZERO']

/*************** MODULE DECLARATIONS ***************/

[INT TOP BLINK EP] [BLINK:=4] [TOP:=30] [EP:=TOP-10]
DECLARE INT$STACK DATA (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
DECLARE BLINK$TIMER DATA (1);
DECLARE SAVHL DATA (0,0);

/******************** PROCEDURES ********************/

DUMMY$HDLR:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE PROVIDES A COMMON EMPTY INTERRUPT   */
/* HANDLER FOR THOSE INTERRUPT LEVELS WHICH SHOULD    */
/* NEVER OCCUR WITH THE CURRENT NPS SYCOR 440 HARD-   */
/* WARE CONFIGURATION. ITS ONLY ACTION IS AN IMMEDI-  */
/* ATE RETURN.                                        */
/* CALLED BY: INTERRUPT$CONTROLLER                    */
/*************************************************************/
    RETURN;
    END DUMMY$HLDR;

DEBUG$HDLR:  PROCEDURE;
/*************************************************************/
/* THIS HANDLER IS INCLUDED TO ALLOW USAGE OF THE     */
/* SYCOR 340B DEBUGGER IN THE SOFT DEBUG MODE WHEN    */
/* RUNNING UNDER MTS. IT WAS DISCOVERED DURING DEVEL- */
/* OPMENT AND TESTING THAT THE DEBUGGER IS LARGELY    */
/* UNRELIABLE IN THE SOFT MODE. THIS IS APPARENTLY    */
/* DUE TO HARDWARE CHANGES IN THE 440 SYSTEM MADE     */
/* AFTER THE DEBUGGER WAS DESIGNED.                   */
/* CALLED BY: INTERRUPT$CONTROLLER                    */
/*************************************************************/
    /* DISPLAY ENVIRONMENT ON DEBUGGER */
    HL=2+SP;
    M(6H)=(A=L); M(7H)=(A=H);
    /* ACKNOWLEDGE DEBUG INTERRUPT */
    OUT([DEBUG$LATCH])=(A=1);
    /* CPU IDLES WHILE WAITING FOR DEBUGGER TO */
    /* INITIATE RESUMPTION OF EXECUTION        */
    DO WHILE [DEBUG$INT$PENDING];
        END;
    END DEBUG$HDLR;

CASSETTE$HDLR:  PROCEDURE;
    OUT([CSST$LATCH])=(A=10H);
    END CASSETTE$HDLR;

PRINTER$HDLR:  PROCEDURE;
    A=IN([PRINTER$LATCH]);
    END PRINTER$HDLR;

TIMER$HDLR:  PROCEDURE;
/*************************************************************/
/* THE TIMER INTERRUPT HANDLER MANAGES THE TWO FUNC-  */
/* TIONS OF MTS WHICH OCCUR AT PERIODIC INTERVALS:    */
/* BLINKING THE TERMINAL CURSORS AND RETURNING CONTROL*/
/* TO THE SYSTEM WHEN A TASK'S TIMESLICE EXPIRES. IN  */
/* ORDER TO KEEP TRACK OF THE TWO INTERVALS INVOLVED, */
/* THE PROCEDURE MAINTAINS TWO COUNTERS. THESE COUN-  */
```

```
/* TERS ARE EACH SET TO AN INITIAL VALUE AND THEN      */
/* DECREMENTED EACH TIME A TIMER INTERRUPT OCCURS.     */
/* THE ACTUAL VALUE CONTAINED IN EITHER COUNTER AT ANY*/
/* INSTANT REPRESENTS THE TIME REMAINING IN THE INTER-*/
/* VAL IN MULTIPLES OF 50MS SINCE THIS IS THE FIXED    */
/* INTERVAL BETWEEN TIMER INTERRUPTS. WHEN THE TASKS   */
/* TIMER COUNTER HAS BEEN DECREMENTED TO ZERO, CONTROL*/
/* IS TRANSFERRED TO THE MONITOR WHERE THE CURRENT     */
/* TASK IS SWAPPED OUT AND A NEW TASK SWAPPED IN.      */
/* WHEN THE BLINK$TIMER COUNTER REACHES ZERO THE       */
/* BLINK$CURSOR PROCEDURE IS CALLED. IN EITHER CASE    */
/* THE TIMER HANDLER RESETS THE COUNTER TO ITS INIT-   */
/* IAL VALUE AND CONTINUES.                            */
/* CALLED BY: INTERRUPT$CONTROLLER                     */
/*******************************************************/
    TASK$TIMER=(A=TASK$TIMER-1);
    BLINK$TIMER=(A=BLINK$TIMER-1);
    IF (A::0) ZERO THEN  /* BLINK INTERVAL EXPIRED */
        DO;
        CALL [BLINK$CURSOR];
        BLINK$TIMER=(A=[BLINK]);
        END;
    OUT([TIMER$LATCH])=(A=0);  /* RESET TIMER */
    IF (A=TASK$TIMER; A::0) ZERO THEN
        DO;  /* TIMESLICE EXPIRED */
        IF (A=>LOCK) !CY THEN
            DO;  /* SWAPPING UNLOCKED */
            BC=10; DE=.INT$STACK([EP]);
            HL=.SWAP$STACK;
            CALL [MOVBUF];
            ENABLE;
            GOTO [MONITOR];
            END
        ELSE TASK$TIMER=(A=1);
        END;
    END TIMER$HDLR;

TERMINAL$HDLR:  PROCEDURE;
/*******************************************************/
/* THIS PROCEDURE PROCESSES THE INTERRUPT GENERATED    */
/* FROM ANY KEY DEPRESSION AT ANY OF THE TERMINALS.    */
/* IT GETS THE TERMINAL IDENTITY AND THE KEYBOARD      */
/* MATRIX CODE AND THEN CALLS TERMINAL$INPUT$CONTROL   */
/* TO PROCESS THE KEY.                                 */
/* OUTPUT: C - MATRIX CODE                             */
/*         E - TERMINAL NUMBER                         */
/* CALLED BY: INTERRUPT$CONTROLLER                     */
/*******************************************************/
    /* READ TERMINAL IDENTITY */
    E=(A=IN([TERMINAL$LATCH]) 03);
    /* WRITE TERMINAL NUMBER BACK OUT TO CAUSE */
    /* THE APPROPRIATE KEYBOARD DATA REGISTER  */
    /* TO BE SELECTED FOR READING.             */
    OUT([TERMINAL$LATCH])=A;
    /* READ THE KEYBOARD MATRIX CODE */
    C=(A=IN([MATRIX$LATCH]));
    /* PROCESS KEY */
    CALL [TERM$INPUT$CTRL];
    END TERMINAL$HDLR;


INTERRUPT$CONTROLLER:  /* MAIN ENTRY INTO INTMOD */
    /* SAVE CURRENT VALUE OF STACK POINTER AND */
    /* ALL REGISTERS IN INT$STACK              */
    SAVHL=HL; STACK=PSW;
    HL=2+SP; PSW=STACK;
    SP=.INT$STACK([TOP]);
    STACK=HL;  /* PUSH CURRENT STACK PTR */
    HL=SAVHL;
    STACK=HL;  /* PUSH ORIGINAL CONTENTS OF HL */
    STACK=DE; STACK=BC; STACK=PSW;
    DO WHILE [INT$PENDING];
```

```
        H=(A=0); L=(A=[LEVEL]);
        DO CASE HL;
         /*  0 */  CALL DUMMY$HDLR;
         /*  1 */  CALL CASSETTE$HDLR;
         /*  2 */  CALL DUMMY$HDLR;
         /*  3 */  CALL DUMMY$HDLR;
         /*  4 */  CALL DUMMY$HDLR;
         /*  5 */  CALL DUMMY$HDLR;
         /*  6 */  CALL PRINTER$HDLR;
         /*  7 */  CALL DUMMY$HDLR;
         /*  8 */  CALL TIMER$HDLR;
         /*  9 */  CALL DUMMY$HDLR;
         /* 10 */  CALL TERMINAL$HDLR;
         /* 11 */  CALL DUMMY$HDLR;
         /* 12 */  CALL DUMMY$HDLR;
         /* 13 */  CALL DUMMY$HDLR;
         /* 14 */  CALL DUMMY$HDLR;
         /* 15 */  CALL DUMMY$HDLR;
         /* 16 */  CALL DEBUG$HDLR;
        END; /* CASE */
    END; /* WHILE */
    /* RESTORE ORIGINAL VALUE OF STACK POINTER AND */
    /* ALL REGISTERS FROM INTS$STACK                */
    PSW=STACK; BC=STACK; DE=STACK; HL=STACK;
    SAVHL=HL; HL=STACK;
    SP=HL; HL=SAVHL;
    ENABLE;
    RETURN;
    /* END INTERRUPTS$CONTROLLER */

EOF
```

```
/*************************************************************/
/*                    MONITOR MODULE                         */
/*************************************************************/
/*                                                           */
/* THE MONITOR MODULE CONTAINS THOSE FUNCTIONS OF MTS        */
/* WHICH DEAL WITH PROCESSOR MANAGEMENT. SUCH FUNC-          */
/* TIONS INCLUDE THE INITIAL PROGRAM LOAD, SYSTEM            */
/* RECOVERY, SCHEDULING, CPU ALLOCATION, AND SWAPPING.       */
/* THE MODULE IS DIVIDED INTO THREE BASIC SUBMODULES.        */
/*                                                           */
/*  (1) UTILITY PROCEDURES                                   */
/*      THIS SUBMODULE CONTAINS GENERAL PURPOSE              */
/*      UTILITY PROCEDURES WHICH PERFORM OPERATIONS          */
/*      FREQUENTLY REQUIRED IN THE MONITOR, INTERRUPT        */
/*      AND SERVICE MODULES.                                 */
/*          * INDEX              * PUT                       */
/*          * INDEX2             * GET                       */
/*          * INDEX4             * MOVBUF                    */
/*          * INDEX8             * MINI$DISK                 */
/*                                                           */
/*  (2) TASK MANAGEMENT                                      */
/*      THIS SUBMODULE CONTAINS THE SCHEDULING, CPU          */
/*      ALLOCATION, AND SWAPPING PROCEDURES. IT ALSO         */
/*      INCORPORATES THE MECHANISM FOR RECORDING THE         */
/*      SYSTEM STATE BLOCK WHEN THE SYSTEM STATE             */
/*      CHANGES, THUS MAKING RECOVERY POSSIBLE.              */
/*      CONTROL PASSES TO THE TASK MANAGEMENT                */
/*      SUBMODULE AFTER MTS HAS BEEN INITIALIZED BY          */
/*      THE IPL SUBMODULE.                                   */
/*          * MONITOR            * SWAP                      */
/*          * BUMPSTASK          * WRITE$REC                 */
/*          * BOOTSTRAP          * RESUME$EXECUTION          */
/*                                                           */
/*  (3) INITIAL PROGRAM LOAD                                 */
/*      THIS SUBMODULE CONTAINS ALL PROCEDURES WHICH         */
/*      DEAL WITH THE LOADING PROCESS AFTER THE MTS          */
/*      OBJECT MODULE HAS BEEN LOADED INTO MEMORY BY         */
/*      THE SYCOR SYSTEM LOADER. THE PRIMARY FUNCTION        */
/*      OF THE IPL SUBMODULE IS SYSTEM INITIALIZATION        */
/*      OR RECOVERY, AS REQUIRED. IN ORDER TO MINI-          */
/*      MIZE THE MEMORY REQUIREMENT OF THE RESIDENT          */
/*      MTS CODE, THIS SUBMODULE WAS WRITTEN AS A            */
/*      STANDALONE PROGRAM LOADED INTO THE USER SWAP         */
/*      AREA. ONCE IPL IS COMPLETE THE PROGRAM MAY           */
/*      BE OVERLAYED BY USER PROGRAMS.                       */
/*          * ABORT$IPL          * READ$DIRECTORY            */
/*          * SEARCH$DIRECTORY   * RECOVER                   */
/*          * INITIALIZE         * MTS$IPL                   */
/*                                                           */
/* THE MONITOR MODULE REQUIRES ACCESS TO SEVERAL             */
/* FILES WHICH RESIDE ON THE MINI-DISK IN SYCOR              */
/* FORMAT. THESE FILES AND THEIR FUNCTION ARE:               */
/*                                                           */
/*  (1) .MTSSWP0,.MTSSWP1,.MTSSWP2,.MTSSWP3                 */
/*      ASSOCIATED WITH EACH OF THE FOUR TERMINAL            */
/*      TASKS IS A FILE USED TO STORE A CORE IMAGE           */
/*      OF THE TASK WHEN IT IS INACTIVE OR BLOCKED.          */
/*      THIS SWAP IMAGE CONSISTS OF A SYSTEM AREA            */
/*      WHERE THE ENVIRONMENT AND VIRTUAL DEVICE             */
/*      CONTROL BLOCK IS STORED, AND A USER AREA             */
/*      CONTAINING THE USER PROGRAM'S MEMORY IMAGE.          */
/*      THE MAXIMUM SWAP IMAGE SIZE IS 48K BYTES.            */
/*      THEREFORE, EACH SWAP FILE SHOULD NORMALLY BE         */
/*      96 MINI-DISK SECTORS LONG. IN THE EVENT THAT         */
/*      MINI-DISK SPACE IS LIMITED AND THE ENTIRE 48K        */
/*      IS NOT REQUIRED FOR USER PROGRAMS, MTS WILL          */
/*      AUTOMATICALLY ADJUST TO ANY FILE SIZE GREATER        */
/*      THAN 16K (32 SECTORS). THE FOLLOWING SYCOR           */
/*      COMMAND MAY BE USED TO CREATE A SWAP FILE:           */
/*              CREATE <FILENAME> N=96                       */
/*      THE SYCOR SYSTEM DOES NOT ALLOW DYNAMIC              */
/*      CHANGES IN FILE SIZE OR ATTRIBUTES; THUS, IN         */
```

```
/*      ORDER TO CHANGE THE FILE SIZE, THE FILE MUST  */
/*      FIRST BE DELETED (DELETE <FILENAME>) AND THEN */
/*      RECREATED WITH THE DESIRED SIZE. IF SWAP      */
/*      FILES SMALLER THAN 48K ARE DESIRED, THE VALUE */
/*      OF N IN THE CREATE COMMAND STRING SHOULD BE   */
/*      TWO TIMES THE REQUIRED PROGRAM SPACE IN       */
/*      KILOBYTES.                                    */
/*      THE FOUR SWAP FILES ARE ONLY REQUIRED WHEN    */
/*      MTS IS ACTUALLY RUNNING, OR BETWEEN RUNS IF   */
/*      RECOVERY WILL BE REQUESTED.                   */
/*                                                    */
/*  (2) .MTSCNFG                                      */
/*      THE VIRTUAL FLOPPY DISK CONFIGURATION FILE    */
/*      PROVIDES THE MAPPING BETWEEN THE VIRTUAL DISK */
/*      NUMBER (0-31) AND THE NAME OF A SYCOR FILE    */
/*      WHICH CONTAINS A FLOPPY DISK IMAGE. THE CON-  */
/*      FIGURATION FILE ALSO CONTAINS THE PROTECTION  */
/*      ATTRIBUTES OF THE VIRTUAL DISKS. WHERE A      */
/*      PHYSICAL FLOPPY DISK HAS A FIXED CAPACITY OF  */
/*      256K BYTES, AN MTS DISK IMAGE MAY HAVE ANY    */
/*      CONVENIENT SIZE. THE SYSTEM ASSUMES THAT THE  */
/*      DISK IMAGE IS STORED SEQUENTIALLY ON THE      */
/*      MINI-DISK STARTING WITH TRACK 0 SECTOR 1 AND  */
/*      PROCEEDING THROUGH 26 SECTORS OF TRACK 0 TO   */
/*      TRACK 1 SECTOR 0, ETC. SPECIFYING A VIRTUAL   */
/*      DISK FILE SMALLER THAN 256K BYTES MEANS THAT  */
/*      FEWER THAN 77 TRACKS WILL BE ADDRESSABLE,     */
/*      WHILE A SIZE GREATER THAN 256K BYTES WILL     */
/*      MAKE MORE THAN 77 TRACKS ADDRESSABLE, UP TO   */
/*      A MAXIMUM OF 256 TRACKS. IN EITHER CASE MTS   */
/*      AUTOMATICALLY ADJUSTS THE UPPER BOUND BASED   */
/*      ON THE FILE SPACE AVAILABLE.                  */
/*      THE CONFIGRATION FILE CONTAINS 32 THIRTEEN    */
/*      BYTE RECORDS IN THE FOLLOWING FORMAT:         */
/*              ----------------------                */
/*              | FILENAME | KEY | P |                */
/*              ----------------------                */
/*               0        7 8  11  12                 */
/*      WHERE 'FILENAME' IS THE 0-8 BYTE NAME OF THE  */
/*      VIRTUAL DISK FILE AS IT APPEARS IN THE SYCOR  */
/*      DIRECTORY; 'KEY' IS A 0-4 BYTE PROTECTION     */
/*      KEY; AND 'P' IS THE PROTECTION ATTRIBUTE OF   */
/*      THE DISK, I.E. 'P' FOR READ/WRITE PROTECTION, */
/*      'R' FOR WRITE PROTECTION ONLY, AND BLANK FOR  */
/*      NO PROTECTION.                                */
/*      THE CONFIGURATION FILE WILL BE UPDATED BY THE */
/*      MTS SYSTEM COMMANDS PROTECT, UNPROTECT, AND   */
/*      RESTRICT. ALTERNATIVELY, IT MAY BE MODIFIED   */
/*      USING SYCOR'S DATA ENTRY FREE FORM MODE. IN   */
/*      THE EVENT THAT .MTSCNFG IS ERRONEOUSLY        */
/*      DELETED, THE FILE MAY BE RECREATED BY USING   */
/*      THE SYCOR COMMAND                             */
/*              RUN RESTORE 2=/CSST                   */
/*      WITH THE CASSETTE LABELED ".MTSCNFG DUMP"     */
/*      MOUNTED ON THE CASSETTE DRIVE.                */
/*                                                    */
/*  (3) .MTSRCVR                                      */
/*      THE RECOVERY FILE CONTAINS A COPY OF THE      */
/*      SYSTEM STATE BLOCK AS OF THE LAST SWAP. THE   */
/*      FILE IS ONLY REQUIRED WHEN MTS IS ACTUALLY    */
/*      RUNNING, OR BETWEEN RUNS IF RECOVERY WILL BE  */
/*      REQUESTED. THE FOLLOWING SYCOR COMMAND IS     */
/*      USED TO CREATE THE FILE:                      */
/*              CREATE .MTSRCVR N=1                   */
/*                                                    */
/******************************************************/
/******************************************************/
```

```
/*************************************************************/
/******************** TASK MANAGEMENT ************************/
/*************************************************************/


/************** INTERMODULE LINKAGE MACROS *****************/

[INT TB M2B] [TB:=1000H] [M2B:=0600H]
[MACRO MCP '1A00H']
[MACRO MTS '1F00H']
[MACRO INDEX '[HEX M2B + 3H]']
[MACRO INDEX2 '[HEX M2B + 0DH]']
[MACRO INDEX8 '[HEX M2B + 24H]']
[MACRO GET '[HEX M2B + 38H]']
[MACRO MINI$DISK '[HEX M2B + 54H]']
[MACRO MTS$MSG '[HEX TB + 837H]']
[MACRO READ$TERMINAL '[HEX TB + 8DCH]']
[MACRO GET$TERM$STATUS '[HEX TB + 2F9H]']

/***************** GENERAL PURPOSE MACROS ******************/

[INT MEM$BASE TOP TIMESLICE]
[MEM$BASE:=4000H]  [TIMESLICE:=4]  [TOP:=20]
[MACRO SSB$BASE 'TASK']
[MACRO SWAP$BASE 'VDC$DRIVE']
[MACRO READ '1']
[MACRO WRITE '2']
[MACRO BUMP '-2']
[MACRO DISK$ERROR '(A::0) !ZERO']
[MACRO HARDWARE$ERROR '8']
[MACRO INPUTS$WAITING '0FFH']
[MACRO IN '[READ]']
[MACRO OUT '[WRITE]']

/****************** MODULE DECLARATIONS ********************/

DECLARE (MONITOR, BOOTSTRAP, RESUME$EXECUTION) LABEL;
DECLARE DIR DATA (0);
DECLARE SAVHL DATA (0,0);
DECLARE I DATA (0);
DECLARE TN DATA (0);

/********************** PROCEDURES *************************/

BUMP$TASK:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE DELETES A TASK FROM THE SYSTEM WHEN */
/* AN IRRECOVERABLE MINI-DISK ERROR OCCURS.           */
/* CALLED BY: SWAP, BOOTSTRAP                         */
/*************************************************************/
    C=[BUMP]; CALL [MTS];
    E=[HARDWARE$ERROR]; CALL [MTS$MSG];
    END BUMP$TASK;

SWAP:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE SWAPS A TERMINAL TASK BETWEEN MEMORY*/
/* AND THE APPROPRIATE MINI-DISK SWAP FILE. THE DIR-  */
/* ECTION OF THE SWAP, I.E. IN OR OUT, IS DETERMINED  */
/* BY THE VALUE OF THE VARIABLE DIR. THE              */
/* PROCEDURE ALSO MODIFIES TCT$STATUS TO REFLECT THE  */
/* CURRENT LOCATION OF THE SWAP IMAGE.                */
/* INPUT: DIR - DIRECTION OF SWAP                     */
/*                1 = IN                              */
/*                2 = OUT                             */
/* CALLED BY: MONITOR                                 */
/*************************************************************/
    /* SET I TO SWAP IMAGE SIZE */
    DE=.TCT$SIZE; A=TASK; CALL [INDEX];
    I=(A=M(HL));
    /* MODIFY TCT$STATUS */
    HL=.TCT$STATUS+BC;
```

```
    M(HL)=(A=M(HL) \\ 0C0H);
    /* SET UP REGISTERS FOR DATA TRANSFER */
    DE=.TCT$BOE; A=TASK; CALL [INDEX2]; CALL [GET];
    DE=.[SWAP$BASE]; A=I;
    DO WHILE (A::0) !ZERO;
        L=(A=DIR); CALL [MINI$DISK];
        IF [DISK$ERROR] THEN
            DO;  /* BUMP TASK OFF SYSTEM */
            CALL BUMP$TASK;
            IF (A=DIR; A::[IN]) ZERO GOTO MONITOR;
            RETURN;
            END;
        BC=BC+1; DE=(HL=200H+DE);
        I=(A=I-1);
    END;  /* WHILE */
    END SWAP;

WRITE$REC:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE COPIES THE CONTENTS OF THE SYSTEM          */
/* STATE BLOCK TO THE RECOVERY FILE. THE VARIABLE            */
/* REC$FILE MUST BE SET TO THE FILE'S SECTOR ADDRESS         */
/* BEFORE CALLING WRITE$REC.                                 */
/* CALLED BY: BOOTSTRAP, RESUME$EXECUTION                    */
/*************************************************************/
    BC=(HL=REC$FILE); DE=.[SSB$BASE];
    L=[WRITE]; CALL [MINI$DISK];
    IF [DISK$ERROR] THEN HALT;
    END WRITE$REC;

MONITOR:
/*************************************************************/
/* THIS ROUTINE IS THE TASK MANAGER OR SCHEDULER             */
/* WHICH CONTROLS THE ALLOCATION OF THE CPU TO COM-          */
/* PETING TERMINAL TASKS.  IT PERFORMS THIS FUNCTION         */
/* BY SEQUENTIALLY SCANNING THE TCT$STATUS BYTE              */
/* ASSOCIATED WITH EACH TERMINAL LOOKING FOR A TASK          */
/* REQUIRING THE CPU.  THE EFFECT PRODUCED IS THAT           */
/* OF A ROUND-ROBIN SCHEDULING ALGORITHM.  WHILE THE         */
/* MONITOR IS LOOPING, IT INITIATES SWAPPING AND             */
/* PRINTING OF SPOOLER FILES AS REQUIRED.                    */
/* CALLED BY: MTS$IPL, MTS (SVCMOD)                          */
/*************************************************************/
    SP=.SYS$STACK([TOP]);  /* SET STACK POINTER */
    /* LOCK OUT SWAPPING */
    LOCK=(A=LOCK \ 01H);
    /* INITIALIZE TEMP TASK COUNTER */
    TN=(A=TASK);
    LOOP:  /* SEARCH FOR READY TASK */
        TN=(A=TN+1,803H); /* INCREMENT TASK NUMBER */
        /* TEST FOR INACTIVE TASK */
        DE=.TCT$STATUS; CALL [INDEX]; A=M(HL);
        IF (A::0) ZERO GOTO LOOP;
        /* TEST BIT 0 - BOOTSTRAP LOAD */
        IF (A=>A) CY THEN
            DO;
            DE=.TCT$STATUS; A=TASK; CALL [INDEX];
            IF (A=<M(HL)) CY
              (B=(A=TN); A=TASK-B) !ZERO THEN
                DO; DIR=(A=[OUT]); CALL SWAP; END;
            TASK=(A=TN);
            GOTO BOOTSTRAP;
            END;
        /* TEST BIT 1 - MCP */
        IF (A=>A) CY THEN
            DO;
            A=TASK; STACK=PSW;  /* SAVE OLD TASK NR */
            TASK=(A=TN); CALL [MCP];
            DE=.TCT$STATUS; A=TN; CALL [INDEX];
            M(HL)=(A=M(HL)  0FDH);
                        /* RESET BIT 1 */
            PSW=STACK; TASK=A;
```

```
                    /* RESTORE OLD TASK NR */
        TN=(A=TN-1,803H); GOTO LOOP;
                    /* CONTINUE WITH TASK AFTER */
                    /* SYSTEM CALL PROCESSED    */
        END;
    /* SKIP BIT 2 - RESERVED FOR CASSETTE */
    A=>A;
    /* SKIP BIT 3 - RESERVED FOR ASYNC */
    A=>A;
    /* SKIP BIT 4 - BLOCKED FOR PRINTER I/O */
    A=>A;
    /* TEST BIT 5 - BLOCKED FOR TERMINAL I/O */
    IF (A=>A) CY THEN
        DO;
        A=TN; CALL [GET$TERM$STATUS];
        IF (A::[INPUT$WAITING]) ZERO THEN
            DO; /* NO LONGER BLOCKED */
            DE=.TCT$STATUS; A=TASK; CALL [INDEX];
            IF (A=<M(HL)) CY
            & (B=(A=TN); A=TASK-B) !ZERO THEN
                DO; DIR=(A=[OUT]); CALL SWAP; END;
            TASK=(A=TN);
            DIR=(A=[IN]); CALL SWAP;
            CALL [READ$TERMINAL];
            SWAP$STACK(1)=A;
                    /* RETURN CHAR REQUESTED */
            DE=.TCT$STATUS; A=TASK; CALL [INDEX];
            M(HL)=(A=M(HL)  0DFH);
                                /* RESET BIT 5 */
            GOTO RESUME$EXECUTION;
            END
        ELSE GOTO LOOP;
        END;
    /* TEST BIT 6 - RESUME EXECUTION - FM DISK */
    IF (A=>A) CY THEN
        DO;
        DE=.TCT$STATUS; A=TASK; CALL [INDEX];
        IF (A=<M(HL)) CY THEN
            DO; DIR=(A=[OUT]); CALL SWAP; END;
        TASK=(A=TN);
        DIR=(A=[IN]); CALL SWAP;
        GOTO RESUME$EXECUTION;
        END;
    /* BIT 7 SET - RESUME EXECUTION - IN MEMORY */
    GOTO RESUME$EXECUTION;
    /* END MONITOR */

BOOTSTRAP:
/*******************************************************/
/* THIS ROUTINE EXAMINES THE DISK MAP FOR THE CURRENT */
/* TASK; DETERMINES THE VIRTUAL DISK ATTACHED TO DRIVE*/
/* A; LOADS THE FIRST 512 BYTES FROM THE DISK INTO     */
/* MEMORY STARTING AT THE BASE OF THE USER SWAP AREA; */
/* AND THEN TRANSFERS CONTROL TO THE CODE JUST LOADED.*/
/*******************************************************/
    /* DETERMINE DISK NR ATTACHED TO DRIVE A */
    DE=.TCT$DN; A=TASK; CALL [INDEX8];
    A=M(HL)  1FH;
    /* DETERMINE BOE FOR DISK */
    DE=.DMT$BOE; CALL [INDEX2]; CALL [GET];
    /* READ FIRST SECTOR ON VIRTUAL DISK */
    DE=[MEM$BASE]; L=[READ]; CALL [MINI$DISK];
    IF [DISK$ERROR] THEN
        DO;  /* BUMP TASK OFF SYSTEM */
        CALL BUMP$TASK;
        CALL WRITE$REC;
        GOTO MONITOR;
        END;
    /* UPDATE SYSTEM STATUS */
    DE=.TCT$STATUS; A=TASK; CALL [INDEX];
    A=M(HL) \ 80H;  /* SET BIT 7 */
    M(HL)=(A=A  0FEH);  /* RESET BIT 0 */
```

```
    CALL WRITE$REC;
    TASK$TIMER=(A=[TIMESLICE]);  /* RESET TASK$TIMER */
    LOCK=(A=LOCK  0FEH);  /* UNLOCK SWAPPING */
    SP=0FEFFH;
    GOTO [MEM$BASE];
    /* END BOOTSTRAP */

RESUME$EXECUTION:
/*************************************************************/
/* THIS ROUTINE TRANSFERS CONTROL BACK TO A USER TASK */
/* WHICH HAS BEEN SWAPPED INTO MEMORY.                       */
/*************************************************************/
    CALL WRITE$REC;
    /* UPDATE SYSTEM STATUS */
    TASK$TIMER=(A=[TIMESLICE]);  /* RESET TASK$TIMER */
    /* RESTORE ORIGINAL VALUE OF STACK POINTER */
    /* AND ALL REGISTERS FROM SWAP$STACK       */
    SP=.SWAP$STACK;
    PSW=STACK; BC=STACK; DE=STACK; HL=STACK;
    SAVHL=HL; HL=STACK;
    SP=HL; HL=SAVHL;  /* RESTORE USER SP */
    STACK=PSW;
    LOCK=(A=LOCK  0FEH);  /* UNLOCK SWAPPING */
    PSW=STACK;
    RETURN; /* RETURNS TO INTERRUPT POINT */
            /* IN USER SWAP IMAGE          */
    /* END RESUME$EXECUTION */


EOF
/*************************************************************/
/****************** UTILITY PROCEDURES ***********************/
/*************************************************************/


INDEX:  PROCEDURE;
/*************************************************************/
/* GIVEN THE BASE ADDRESS OF A BYTE VECTOR AND AN      */
/* INDEX VALUE, THIS PROCEDURE CALCULATES THE ADDRESS */
/* OF THE INDEXED ENTRY.                               */
/* INPUT:  A - INDEX VALUE (USUALLY TASK)              */
/*        DE - BASE ADDRESS OF VECTOR                  */
/* OUTPUT: BC - INDEX VALUE                            */
/*         DE - SAME AS INPUT                          */
/*         HL - CALCULATED ADDRESS OF INDEXED ENTRY    */
/* CALLED BY: SWAP, MONITOR, VAL$DISK, CLEAR$FLAG,     */
/*            ATTACH, LOGIN, QUIT, SIZE, TERM$BLOCK,   */
/*            SELECT$DRIVE                             */
/*************************************************************/
    B=0; C=A;
    HL=BC+DE;
    END INDEX;

INDEX2:  PROCEDURE;
/*************************************************************/
/* GIVEN THE BASE ADDRESS OF AN ADDRESS VECTOR AND AN */
/* INDEX VALUE, THIS PROCEDURE CALCULATES THE ADDRESS */
/* OF THE LOW ORDER BYTE OF THE INDEXED ENTRY.         */
/* INPUT:  A - INDEX VALUE (USUALLY TASK)              */
/*        DE - BASE ADDRESS OF VECTOR                  */
/* OUTPUT: BC - CALCULATED OFFSET = 2 * INDEX VALUE    */
/*         DE - SAME AS INPUT                          */
/*         HL - CACULATED ADDRESS OF INDEXED ENTRY     */
/* CALLED BY: SWAP, BOOTSTRAP, SIZE, SELECT$DRIVE      */
/*************************************************************/
    B=0; C=(A=<<A);
    HL=BC+DE;
    END INDEX2;

INDEX4:  PROCEDURE;
/*************************************************************/
/* INPUT:  A - INDEX VALUE                             */
```

```
/*         DE - BASE ADDRESS OF VECTOR                          */
/* OUTPUT: BC - CALCULATED OFFSET = 4 * INDEX VALUE             */
/*         DE - SAME AS INPUT                                   */
/*         HL - CALCULATED ADDRESS OF INDEXED ENTRY             */
/* CALLED BY: VAL$KEY                                           */
/*--------------------------------------------------------------*/
    B=0; C=(A=<<(A=<<A));
    HL=BC+DE;
    END INDEX4;

INDEX8:  PROCEDURE;
/*--------------------------------------------------------------*/
/* INPUT:  A - INDEX VALUE                                      */
/*        DE - BASE ADRESS OF VECTOR                            */
/* OUTPUT: BC - CALCULATED OFFSET = 8 * INDEX VALUE             */
/*         DE - SAME AS INPUT                                   */
/*         HL - CALCULATED ADDRESS OF INDEXED ENTRY             */
/* CALLED BY: BOOTSTRAP, VAL$DRIVE, CLEAR$DM, ATTACH,           */
/*            SELECT$DRIVE                                      */
/*--------------------------------------------------------------*/
    B=0; C=(A=<<(A=<<(A=<<A)));
    HL=BC+DE;
    END INDEX8;

PUT:  PROCEDURE;
/*--------------------------------------------------------------*/
/* STORE A TWO BYTE ADDRESS IN A SPECIFIED VECTOR.              */
/* INPUT: DE - ADDRESS TO BE STORED                             */
/*        HL - BASE ADDRESS OF VECTOR                           */
/* OUTPUT: BC - UNCHANGED                                       */
/*         DE - SAME AS INPUT                                   */
/*         HL - BASE ADDRESS + 1                                */
/* CALLED BY: RECOVER, INITIALIZE                               */
/*--------------------------------------------------------------*/
    M(HL)=E; HL=HL+1; M(HL)=D;
    END PUT;

GET:  PROCEDURE;
/*--------------------------------------------------------------*/
/* FETCH A TWO BYTE ADDRESS FROM A SPECIFIED VECTOR.            */
/* INPUT: HL - BASE ADDRESS OF VECTOR                           */
/* OUTPUT: BC - ADDRESS FETCHED                                 */
/*         DE - ADDRESS FETCHED                                 */
/*         HL - BASE ADDRESS + 1                                */
/* CALLED BY: SWAP, BOOTSTRAP, SIZE, SELECT$DRIVE               */
/*--------------------------------------------------------------*/
    E=M(HL); HL=HL+1; D=M(HL); BC=DE;
    END GET;

MOVBUF:  PROCEDURE;
/*--------------------------------------------------------------*/
/* THIS IS A GENERAL PURPOSE UTILITY PROCEDURE WHICH            */
/* MOVES A SPECIFIED NUMBER OF BYTES FROM A SOURCE              */
/* BUFFER TO A DESTINATION BUFFER.                              */
/* INPUT: BC - NUMBER OF BYTES TO BE MOVED                      */
/*        DE - BASE ADDRESS OF SOURCE BUFFER                    */
/*        HL - BASE ADDRESS OF DESTINATION BUFFER               */
/* CALLED BY: MTS$IPL, ABORT$IPL, SEARCH$DIRECTORY,             */
/*            LOGIN, READ$FLOPPY, WRITE$FLOPPY                  */
/*--------------------------------------------------------------*/
    REPEAT;
        M(HL)=(A=M(DE));
        HL=HL+1; DE=DE+1;
    UNTIL (BC=BC-1; A=0; A::C) ZERO (A::B) ZERO;
    END MOVBUF;

MINI$DISK:  PROCEDURE;
/*--------------------------------------------------------------*/
/* THIS PROCEDURE TRANSFERS A SPECIFIED 512 BYTE                */
/* BUFFER BETWEEN MEMORY AND THE MINI-DISK. THE DIR-            */
/* ECTION OF THE TRANSFER IS INDICATED BY THE OP CODE.          */
/* ERROR PROCESSING IS LIMITED TO CHECKING THE STATUS           */
```

```
/* RETURNED BY THE MINI-DISK CONTROLLER FOR 'NORMAL   */
/* COMPLETION.' WHEN ANY OTHER STATUS IS RETURNED THE */
/* ROUTINE DISPLAYS THE MESSAGE 'HARDWARE ERROR' ON   */
/* THE TERMINAL CURRENTLY ALLOCATED THE CPU AND       */
/* ABORTS THE OPERATION.                              */
/* INPUT: BC - MINI-DISK SECTOR NUMBER                */
/*        DE - DMA BUFFER BASE ADDRESS                */
/*         L - OPERATION CODE:                        */
/*               1 = READ                             */
/*               2 = WRITE                            */
/*               3 = WRITE/VERIFY                     */
/* OUTPUT: A - RETURNS 00H IF COMPLETION NORMAL,      */
/*             OTHERWISE RETURNS FFH                  */
/*        BC - SAME AS INPUT                          */
/*        DE - SAME AS INPUT                          */
/* CALLED BY: READ$DIRECTORY, RECOVER, INITIALIZE,    */
/*            WRITE$REC,SWAP,BOOTSTRAP, READ$BUF,     */
/*            WRITE$BUF                               */
/******************************************************/
    [MACRO ABNORMAL$COMPLETION '0FFH']
    [MACRO HARDWARE$ERROR '3']
    [MACRO MTS$MSG '1837H']
    /* SET DCB CHECKSUM IN LOCN 4CH */
    M(4CH)=(A=0\B,\C,\D,\E,\L);
    /* SET SECTOR NUMBER IN DISK CONTROL BLOCK */
    M(45H)=(A=B); M(44H)=(A=C);
    /* SET DMA BUFFER ADDRESS IN DISK CONTROL BLOCK */
    M(43H)=(A=D); M(42H)=(A=E);
    /* INITIATE OPERATION */
    M(40H)=(A=L);
    /* WAIT UNTIL OPERATION COMPLETE */
    REPEAT;
        A=M(41H);
    UNTIL (A::0) !ZERO;
    /* TEST A FOR COMPLETION STATUS */
    IF (M(41H)=(A=A-1)) ZERO RETURN;  /* NORMAL COMPLETION */
    M(41H)=(A=0);
    E=[HARDWARE$ERROR];
    CALL [MTS$MSG];
    A=[ABNORMAL$COMPLETION];
    END MINI$DISK;


EOF
/******************************************************/
/***************** INITIAL PROGRAM LOAD ***************/
/******************************************************/


/************** INTERMODULE LINKAGE MACROS *************/

[INT MB M2B SB TB]
[MB:=0300H] [M2B:=0600H] [SB:=1F00H] [TB:=1000H]
[MACRO MONITOR '[HEX MB + 8FH]']
[MACRO MOVBUF '[HEX M2B + 41H]']
[MACRO PUT '[HEX M2B + 31H]']
[MACRO MINI$DISK '[HEX M2B + 54H]']
[MACRO MTS '[HEX SB + 0]']
[MACRO MTS$MSG '[HEX TB + 837H]']
[MACRO CLEAR$STATUS$LINE '[HEX TB + 827H]']
[MACRO TERMINAL$STATUS '[HEX TB + 8D2H]']
[MACRO READ$TERMINAL '[HEX TB + 8DCH]']
[MACRO WRITE$TERMINAL '[HEX TB + 93CH]']
[MACRO REC$FILE '3E91H']
[MACRO TCT$EOE '3EC5H']
[MACRO CNFG$FILE '3E93H']
[MACRO DMT$FLAG '3ECDH']
[MACRO DMT$BOE '3EEDH']
[MACRO DMT$EOE '3F2DH']
[MACRO DMT$KEY '3F6DH']
[MACRO SYS$STACK '3C7AH']
```

```
/*************** GENERAL PURPOSE MACROS ***************/

[INT MEM$BASE DIR$BASE SSB$BASE TOP]
[MEM$BASE:=5000H]  [SSB$BASE:=3E90H]  [TOP:=20]
[DIR$BASE:=MEM$BASE + 200H]
[MACRO READ '1']
[MACRO DISK$ERROR '(A::0) !ZERO']
[MACRO DISABLE$TIMER 'OUT(2)=(A=1)']
[MACRO CLEAR$CSST$INT 'OUT(85H)=(A=10H)']
[MACRO CLEAR$PRNT$INT 'A=IN(8AH)']
[MACRO RESET$TIMER 'OUT(2)=(A=0)']

/***************** MODULE DECLARATIONS *****************/

DECLARE I BYTE;
DECLARE MAX(2) BYTE;
    /* ADDRESS OF LAST ENTRY + 1 IN DIRECTORY IMAGE */
DECLARE REC$NAME(9) BYTE INITIAL ('.MTSRCVR$');

/******************** PROCEDURES ***********************/

ABORT$IPL:  PROCEDURE(MSG);
/*************************************************************/
/* WHENEVER A CONDITION OCCURS DURINC THE IPL PROCESS */
/* WHICH PREVENTS NORMAL COMPLETION OF THE IPL THIS   */
/* PROCEDURE IS CALLED TO TERMINATE EXECUTION AND     */
/* DISPLAY AN ERROR MESSAGE AT TERMINAL 0.            */
/* INPUT: MSG - BASE ADDRESS OF ERROR MESSAGE         */
/*              TERMINATED BY '$'                     */
/* CALLED BY: READ$DIRECTORY, INITIALIZE, RECOVER,    */
/*************************************************************/
    DECLARE ABORT$MSG DATA ('IPL ABORTED - ');
    /* DISPLAY 'IPL ABORTED' AT TERMINAL 0 */
    BC=14; DE=.ABORT$MSG; HL=0700H;
    CALL [MOVBUF];
    HL=MSG; A='$';
    DO C=0 BY C=C+1 WHILE (A::M(HL)) !ZERO;
        HL=HL+1;  /* COUNT CHARS IN MSG */
    END;
    B=0; DE=(HL=MSG); HL=070EH;
    CALL [MOVBUF];  /* DISPLAY MSG AT TERMINAL 0 */
    HALT;
    END ABORT$IPL;

READ$DIRECTORY:  PROCEDURE;
/*************************************************************/
/* DURING THE INITIALIZATION PROCESS IT IS NECESSARY  */
/* TO DETERMINE THE SECTOR NUMBERS  OF SEVERAL SYSTEM */
/* FILES WHICH RESIDE ON THE MINI-DISK IN SYCOR FORMAT.*/
/* MULTIPLE DIRECTORY SEARCHES COULD LEAD TO REPEATEDLY*/
/* READING THE SAME BLOCK OF MINI-DISK SECTORS. TO    */
/* ELIMINATE MOST OF THESE UNNECESSARY READ OPERATIONS */
/* THIS PROCEDURE READS THE ENTIRE SYCOR DIRECTORY    */
/* INTO MEMORY AT ONE TIME, THUS REDUCING THE OVERHEAD */
/* INVOLVED IN MULTIPLE SEARCHES.                     */
/* CALLED BY: MTS$IPL                                 */
/*************************************************************/
    [INT SYCOR$DIR$BASE]  /* SYCOR DIRECTORY BASE */
    [SYCOR$DIR$BASE:=20H] /* SECTOR NUMBER        */
    DECLARE MSG DATA ('CANNOT READ DIRECTORY$');
    /* SET UP REGISTERS FOR DISK READ */
    BC=[HEX SYCOR$DIR$BASE];
    DE=[HEX DIR$BASE];
    /* READ NUMBER OF SECTORS INDICATED */
    /* IN DIRECTORY HEADER RECORD       */
    REPEAT;
        L=[READ]; CALL [MINI$DISK];
        IF [DISK$ERROR] CALL ABORT$IPL(.MSG);
        DE=(HL=200H+DE); BC=BC+1;
    UNTIL (A=M([HEX DIR$BASE+0AH])); A::C) CY;
    /* CALCULATE ADDRESS OF LAST ENTRY + 1 IN IMAGE */
    B=[HEX SYCOR$DIR$BASE-1];
```

```
    A=A-B;  /* A = NR SECTORS IN DIRECTORY */
    D=(A=<A);
    E=0;  /* DE = NR SECTORS * 512 */
    MAX=(HL=[HEX DIR$BASE+1]+DE);
    END READ$DIRECTORY;

SEARCH$DIRECTORY:  PROCEDURE;
/*************************************************************/
/* GIVEN THE BASE ADDRESS OF A VECTOR CONTAINING THE  */
/* NAME OF A FILE IN SYCOR FORMAT, THIS ROUTINE WILL  */
/* SEARCH THE DIRECTORY IMAGE READ INTO MEMORY BY     */
/* READ$DIRECTORY. IF THE FILE ENTRY IS FOUND, THE    */
/* BOE AND EOE VALUES ARE RETURNED.                   */
/* INPUT: DE - BASE ADDRESS OF FILENAME VECTOR        */
/*             ASSUMED TO BE 8 BYTES LONG             */
/* OUTPUT: BC - BOE OR FFFFH IF FILE NOT FOUND        */
/*         DE - EOE OR FFFFH IF FILE NOT FOUND        */
/* CALLED BY: INITIALIZE, RECOVER                     */
/*************************************************************/
    DECLARE LOOP LABEL;
    /* MOVE FILENAME TO LAST ENTRY + 1 */
    BC=8; HL=MAX; CALL [MOVBUF];
    DE=[HEX DIR$BASE+41H]; /* ADDRESS OF FIRST ENTRY */
    LOOP:  /* ADVANCE TO NEXT ENTRY */
        STACK=DE; HL=MAX; B=8;
        REPEAT;  /* COMPARE CHAR BY CHAR */
            IF (A=M(DE); A::M(HL)) ZERO
             \ (IF (A::0) ZERO  (A=M(HL)-20H) ZERO
                THEN CY=1 ELSE CY=0) CY
            THEN  /* CHAR MATCH */
                DO;
                DE=DE+1; HL=HL+1;
                END
            ELSE  /* NON-MATCH */
                DO; DE=STACK;
                DE=(HL=40H+DE);
                GOTO LOOP;
                END;
        UNTIL (B=B-1) ZERO;
    SP=(HL=2+SP); /* CLEAR STACK */
    /* FALLING THRU LOOP MEANS NAMES MATCH,        */
    /* MUST TEST FOR SUCCESS OR FAILURE OF SEARCH */
    IF (A=MAX(1); A::D) CY  /* X::Y=CY => X<Y */
    \(IF ZERO  (A=MAX(0); A::E) CY THEN CY=1 ELSE CY=0)
        CY THEN
        DO;  /* SEARCH FAILED */
        BC=0FFFFH; DE=BC;
        END
    ELSE DO;  /* SEARCH SUCCESSFUL */
        HL=3+DE;
        C=M(HL); HL=HL+1;
        B=M(HL); HL=HL+1;
        E=M(HL); HL=HL+1;
        D=M(HL);
        END;
    END SEARCH$DIRECTORY;

RECOVER:  PROCEDURE;
/*************************************************************/
/* MTS HAS BEEN DESIGNED SO THAT THE SYSTEM STATE AT  */
/* ANY INSTANT IS DEFINED BY A COMPACT, CONTIGUOUS    */
/* GROUP OF BYTES KNOWN AS THE SYSTEM STATE BLOCK.    */
/* EACH TIME THAT SWAPPING OCCURS THE SSB IS WRITTEN  */
/* TO THE MINI-DISK FILE .MTSRCVR. IF THE TASK JUST   */
/* SWAPPED IN CAUSES A SYSTEM CRASH, RECOVERY IS      */
/* ACCOMPLISHED BY REBOOTING MTS AND ANSWERING 'Y' TO */
/* THE RECOVERY QUERY. MTS WILL READ .MTSRCVR BACK    */
/* INTO THE SSB, DELETE THE OFFENDING TASK, AND       */
/* CONTINUE WITH THE NEXT READY TASK.                 */
/* NOTE: THIS PROCEDURE USES THE FACT THAT THE BOE    */
/*       AND EOE VALUES RETURNED BY SEARCH$DIRECTORY  */
/*       ARE EQUAL FOR A SINGLE-SECTOR FILE.          */
```

```
/* CALLED BY: MTS$IPL                                              */
/*****************************************************************/
    [MACRO BUMP '-2']
    [MACRO OUT$OF$BOUNDS '7']
    /* FIND MINI-DISK SECTOR ADDRESS OF .MTSRCVR */
    DE=.REC$NAME;
    CALL SEARCH$DIRECTORY;
    IF (A=B; A::0FFH) ZERO CALL ABORT$IPL(.REC$NAME);
    /* READ .MTSRCVR INTO SSB */
    HL=[REC$FILE]; CALL [PUT];
    DE=[SSB$BASE];
    L=[READ]; CALL [MINISDISK];
    IF [DISK$ERROR] CALL ABORT$IPL(.REC$NAME);
    /* DELETE TASK CAUSING CRASH */
    C=[BUMP];
    CALL [MTS];
    E=[OUT$OF$BOUNDS];
    CALL [MTS$MSG];
    END RECOVER;

INITIALIZE:  PROCEDURE;
/*****************************************************************/
/* THE SYSTEM STATE BLOCK CONSISTS OF THREE SETS OF              */
/* VARIABLES: SYSTEM CONTROL, TASK CONTROL TABLE, AND            */
/* THE DISK MAP TABLE. THE OBJECT MODULE GENERATED BY            */
/* THE ML80 COMPILER CONTAINS INITIAL VALUES FOR                 */
/* SYSTEM CONTROL VARIABLES AND MOST OF THE TCT. IN              */
/* ORDER TO INITIALIZE THE REST OF THE TCT IT IS                 */
/* NECESSARY TO SEARCH THE SYCOR DIRECTORY IMAGE                 */
/* COPIED INTO MEMORY BY READ$DIRECTORY FOR BOE AND              */
/* EOE VALUES FOR THE RECOVERY FILE AND ALL FOUR SWAP            */
/* FILES. TO INITIALIZE THE DMT THE VIRTUAL DISK                 */
/* CONFIGURATION FILE, .MTSCNFG, MUST BE READ INTO               */
/* MEMORY AND BOE AND EOE VALUES FOR THE VIRTUAL DISK            */
/* FILES EXTRACTED FROM THE SYCOR DIRECTORY IMAGE.               */
/* THE PROTECTION ATTRIBUTES FOR EACH VIRTUAL DISK               */
/* ARE ALSO COPIED INTO THE DMT FROM .MTSCNFG.                   */
/* NOTE: THIS PROCEDURE USES THE FACT THAT THE BOE               */
/*       AND EOE VALUES RETURNED BY SEARCH$DIRECTORY             */
/*       ARE EQUAL FOR A SINGLE-SECTOR FILE.                     */
/* CALLED BY: MTS$IPL                                            */
/*****************************************************************/
    DECLARE ENTRY$BASE(2) BYTE;
    DECLARE CNFG$NAME(9) BYTE INITIAL ('.MTSCNFG$');
    DECLARE SWAP$NAME(9) BYTE INITIAL ('.MTSSWP0$');
    DECLARE SYSDISK(9) BYTE INITIAL ('SYS DISK$');
    /* SET UP RECOVERY FILE */
    DE=.REC$NAME; CALL SEARCH$DIRECTORY;
    IF (A=B; A::0FFH) ZERO CALL ABORT$IPL(.REC$NAME);
    HL=[REC$FILE]; CALL [PUT];
    /* SET UP TASK CONTROL BLOCK IN SSB */
    I=(A=4); STACK=(HL=[TCT$EOE]);
    REPEAT;
        DE=.SWAP$NAME; CALL SEARCH$DIRECTORY;
        IF (A=B; A::0FFH) ZERO
            CALL ABORT$IPL(.SWAP$NAME);
        /* CHECK THAT SWAP FILE AT LEAST 16K */
        L=(A=!C,+1); H=(A=!B,++0);
        HL=HL+DE;
        IF (A=L; A::31) CY CALL ABORT$IPL(.SWAP$NAME);
        HL=STACK; CALL [PUT];
        DE=BC; BC=-9;
        HL=HL+BC; CALL [PUT];
        BC=9; STACK=(HL=HL+BC);
        SWAP$NAME(7)=(A=SWAP$NAME(7)+1);
    UNTIL (I=(A=I-1)) ZERO;
    SP=(HL=2+SP);  /* CLEAR STACK */
    /* SET UP DISK MAP TABLE IN SSB */
    DE=.CNFG$NAME; CALL SEARCH$DIRECTORY;
    IF (A=B; A::0FFH) ZERO CALL ABORT$IPL(.CNFG$NAME);
    HL=[CNFG$FILE]; CALL [PUT];
    /* READ CONFIGURATION FILE INTO MEMORY */
```

```
    DE=[MEM$BASE]; L=[READ]; CALL [MINI$DISK];
    IF [DISK$ERROR]  CALL ABORT$IPL(.CNFG$NAME);
    I=(A=0); ENTRY$BASE=(HL=(DE=[HEX MEM$BASE + 2]));
    REPEAT;  /* STEP THRU CONFIGURATION FILE */
        CALL SEARCH$DIRECTORY;
        IF (A=B; A::0FFH) !ZERO THEN
            DO;  /* VIRTUAL DISK (I) EXISTS */
            STACK=BC; B=0; C=(A=<<I);
            HL=[DMT$EOE]+BC; CALL [PUT];
            DE=STACK; HL=[DMT$BOE]+BC; CALL [PUT];
            BC=8; DE=(HL=ENTRY$BASE+BC);
            B=0; C=(A=<<<(A=<<I));
            HL=[DMT$KEY]+BC;
            DO B=0 BY B=B+1 WHILE (A=B-4) !ZERO;
                M(HL)=(A=M(DE));
                DE=DE+1; HL=HL+1;
            END;
            B=0; C=(A=I);
            HL=[DMT$FLAG]+BC;
            IF (A=M(DE); A::(B='R')) ZERO THEN
                M(HL)=0DH
            ELSE DO;
                IF (A::(B='P')) ZERO THEN M(HL)=05H
                ELSE M(HL)=01H;
                END;
            END;
        DE=(HL=ENTRY$BASE+(BC=13));
        ENTRY$BASE=HL;
    UNTIL (I=(A=I+1); A::32) ZERO;
    /* CHECK THAT DISK 0 EXISTS */
    HL=[DMT$FLAG];
    IF (A=M(HL)  01) ZERO CALL ABORT$IPL(.SYSDISK);
    END INITIALIZE;

MTS$IPL:
/*************************************************************/
/* THIS ROUTINE IS THE INITIAL ENTRY POINT INTO MTS.  */
/* THE SYCOR 440 LOADER TRANSFERS CONTROL HERE AFTER  */
/* THE SYSTEM OBJECT  MODULE HAS BEEN LOADED. DURING  */
/* IPL ALL PERIPHERAL DEVICES ARE RESET, THEN MTS     */
/* READS THE SYCOR DIRECTORY INTO MEMORY, AND ASKS    */
/* THE OPERATOR AT TERMINAL 0 WHETHER RECOVERY IS     */
/* REQUIRED. IF THE ANSWER IS 'Y' THEN THE PROCEDURE  */
/* RECOVER IS CALLED TO READ THE FILE .MTSRCVR INTO   */
/* THE SYSTEM STATE BLOCK. OTHERWISE THE PROCEDURE    */
/* INITIALIZE IS CALLED TO BUILD AN SSB FROM INFOR-   */
/* MATION CONTAINED IN THE SYCOR DIRECTORY IMAGE AND  */
/* THE FILE .MTSCNFG. ONCE IPL IS COMPLETE CONTROL IS */
/* TRANSFERRED TO THE PROCESSOR MANAGEMENT SUBMODULE  */
/* WHICH WILL CONTROL ALL SUBSEQUENT PROCESSING.      */
/* CALLED BY: SYCOR SYSTEM LOADER                     */
/*************************************************************/
    DECLARE (WAIT, TEST) LABEL;
    DECLARE IPL$MSG(15) BYTE INITIAL ('RECOVERY? (Y/N)');
    /* SET STACK POINTER */
    DE=[TOP];
    SP=(HL=[SYS$STACK] + DE);
    /* CLEAR PERIPHERAL INTERRUPTS */
    [DISABLE$TIMER];
    [CLEAR$CSST$INT];
    [CLEAR$PRNT$INT];
    /* CLEAR STATUS LINE ON ALL TERMINALS */
    DO I=(A=0) BY I=(A=I+1) WHILE (A=I; A::4) !ZERO;
        CALL [CLEAR$STATUS$LINE];
    END;
    /* READ SYCOR DIRECTORY INTO MEMORY */
    CALL READ$DIRECTORY;
    /* DISPLAY IPL$MSG AT TERMINAL 0 */
    BC=15; DE=.IPL$MSG; HL=0700H;
    CALL [MOVBUF];
    /* ENABLE INTERRUPTS SO TERMINAL MODULE MAY */
    /* BE USED TO PROCESS REPLY TO IPL$MSG      */
```

```
    ENABLE;
    [RESETSTIMER];
    /* PROCESS OPERATOR'S REPLY  */
    WAIT: REPEAT;
            CALL [TERMINAL$STATUS];
        UNTIL (A::0) !ZERO;
        CALL [READSTERMINAL];
        I=A;
        IF (A::0DH) ZERO GOTO TEST;
        REPEAT;
            CALL [READSTERMINAL];
        UNTIL (A::0DH) ZERO;
    TEST:
        IF (A=I; A::(B='Y')) ZERO
         \ (A::(B=79H)) ZERO
        THEN CALL RECOVER
        ELSE DO;
            IF (A::(B='N')) !ZERO
              (A::(B=6EH)) !ZERO
            THEN
                DO; E='?';
                CALL [WRITESTERMINAL];
                GOTO WAIT;
                END
            ELSE
                DO;
                CALL INITIALIZE;
                A=0; CALL [CLEARSSTATUSSLINE];
                END;
            END;
    GOTO [MONITOR];
    /* END MTS$IPL */

EOF
```

```
/*************************************************************/
/*                      SERVICE MODULE                       */
/*************************************************************/
/*                                                           */
/* THIS MODULE    PROVIDES THE INTERFACE BETWEEN THE         */
/* USER AND ALL SYSTEM SERVICES.  THESE SERVICES FALL        */
/* GENERALLY INTO TWO CATEGORIES:                            */
/*                                                           */
/*  (1) SYSTEM CALLS - THOSE FUNCTIONS REQUIRED TO           */
/*      ESTABLISH THE DESIRED VIRTUAL MACHINE                */
/*      ENVIRONMENT.                                         */
/*                                                           */
/*  (2) SERVICE CALLS - THOSE FUNCTIONS REQUIRED TO          */
/*      ACCESS THE VIRTUAL DEVICES PROVIDED BY THE           */
/*      VIRTUAL MACHINE ENVIRONMENT.                         */
/*                                                           */
/* SYNTACTICALLY THE TWO TYPES OF PROCEDURE CALL ARE         */
/* IDENTICAL, I.E.                                           */
/*          VALUE = MTS(FID,PARM).                           */
/* EACH CALL TAKES TWO ARGUMENTS, FID IN REGISTER C          */
/* AND PARM IN REGISTERS DE; AND RETURNS A VALUE             */
/* IN THE A REGISTER. THE FORM OF THE ARGUMENTS AND          */
/* THE SIDE EFFECTS ASSOCIATED WITH EACH DIFFERENT           */
/* FUNCTION ARE DISCUSSED IN MORE DETAIL IN THE MTS          */
/* USER'S MANUAL. NOTE HOWEVER THAT THE ARGUMENT REG-        */
/* ISTER ASSIGNMENTS CONFORM TO THE PL/M CONVENTION          */
/* FOR PASSING PARAMETERS. THE FOLLOWING TABLE SUM-          */
/* MARIZES THE ARGUMENT OPTIONS AND CORRESPONDING            */
/* RETURNED VALUES.                                          */
/*                                                           */
/*   FID    NAME              PARM         VALUE            */
/*   ---    ----              ----         -----            */
/*   ------------ SYSTEM CALLS -------------                 */
/*    0   ATTACH              LIST         ERROR            */
/*    1   DISPLAY$MSG         ERROR        NONE             */
/*    2   LOGIN               LIST         ERROR            */
/*    3   PROTECT             LIST         ERROR            */
/*    4   QUIT                NONE         NONE             */
/*    5   RESTRICT            LIST         ERROR            */
/*    6   SIZE                SIZE         ERROR            */
/*    7   UNPROTECT           LIST         ERROR            */
/*   ------------ SERVICE CALLS -------------                */
/*    8   TERMINAL$STATUS     NONE         TRUE/FALSE       */
/*    9   READ$TERMINAL       NONE         CHARACTER        */
/*   10   WRITE$TERMINAL      CHARACTER    NONE             */
/*   11   WRITE$PRINTER       CHARACTER    ERROR            */
/*   12   SELECT$DRIVE        DRIVE NR     ERROR            */
/*   13   SET$DMA             DMA ADDRESS  ERROR            */
/*   14   SET$TRACK           TRACK NR     NONE             */
/*   15   SET$SECTOR          SECTOR NR    ERROR            */
/*   16   READ$FLOPPY         NONE         ERROR            */
/*   17   WRITE$FLOPPY        NONE         ERROR            */
/*                                                           */
/* IN THIS TABLE THE ENTRY 'LIST' UNDER PARM INDICATES       */
/* THAT THE ACTUAL ARGUMENT IS THE ADDRESS OF A LIST         */
/* OF REQUIRED PARAMETERS. WHEN 'NONE' APPEARS UNDER         */
/* THE SAME HEADING, IT MEANS THAT THE PARM ARGUMENT         */
/* IS NOT REQUIRED. IF 'NONE' APPEARS UNDER VALUE THE        */
/* ERROR CODE RETURNED IS ALWAYS ZERO. THE ENTRY             */
/* 'ERROR' INDICATES THAT AN ERROR CODE IS RETURNED.         */
/* THESE CODES ARE DEFINED AS FOLLOWS:                       */
/*                                                           */
/*        CODE              ERROR                          */
/*        ----              -----                          */
/*          0          OPERATION SUCCESSFUL                */
/*          1          INVALID COMMAND                     */
/*          2          DISK NOT AVAILABLE                  */
/*          3          DISK IN USE                         */
/*          4          DISK NUMBER ERROR                   */
/*          5          KEY ERROR                           */
/*          6          DRIVE LETTER ERROR                  */
/*          7          OUT OF BOUNDS                       */
```

```
/*          8           HARDWARE ERROR                          */
/*          9           PROTECTION ERROR                        */
/*         10           DRIVE NOT AVAILABLE                     */
/*                                                              */
/* THE SERVICE MODULE IS DIVIDED INTO THREE BASIC               */
/* SUBMODULES:                                                  */
/*                                                              */
/*  (1) USER INTERFACE                                          */
/*      THIS SUBMODULE CONTAINS THE ENTRY POINT INTO            */
/*      THE SERVICE MODULE FROM MCP AND USER PRO-               */
/*      GRAMS. IT IS HERE THAT THE SERVICE REQUEST              */
/*      IS INTERPRETED AND THE APPROPRIATE SERVICE              */
/*      ROUTINE CALLED FOR EXECUTION.                           */
/*                                                              */
/*  (2) SYSTEM CALLS                                            */
/*      THIS SUBMODULE CONTAINS THE SERVICE ROUTINES            */
/*      AND SUPPORTING PROCEDURES WHICH CREATE OR               */
/*      MODIFY THE USER'S VIRTUAL ENVIRONMENT.                  */
/*      THESE ROUTINES ARE INVOKED BY MCP IN RESPONSE           */
/*      TO SYSTEM COMMANDS ENTERED AT A TERMINAL.               */
/*      THEY MAY ALSO BE ACCESSED BY USER PROGRAMS              */
/*      DIRECTLY THROUGH THE USER INTERFACE SUBMODULE.          */
/*                                                              */
/*  (3) SERVICE CALLS                                           */
/*      THIS SUBMODULE CONTAINS SERVICE ROUTINES AND            */
/*      SUPPORTING PROCEDURES WHICH ALLOW A USER                */
/*      PROGRAM TO ACCESS A VIRTUAL TERMINAL OR                 */
/*      VIRTUAL FLOPPY DISK.                                    */
/*                                                              */
/****************************************************************/
/****************************************************************/


/****************************************************************/
/******************** USER INTERFACE ********************/
/****************************************************************/


/************* INTERMODULE LINKAGE MACROS *************/

[INT MB TB M2B] [M2B:=0600H] [MB:=0300H] [TB:=1000H]
[INT S2B S3B]  [S2B:=2200H]  [S3B:=2600H]
[MACRO MONITOR '[HEX MB +8FH]']
[MACRO INDEX '[HEX M2B + 3H]']
[MACRO ATTACH '[HEX S2B + 14AH]']
[MACRO LOGIN '[HEX S2B + 208H]']
[MACRO PROTECT '[HEX S2B + 25CH]']
[MACRO QUIT '[HEX S2B + 261H]']
[MACRO BUMP '[HEX S2B + 293H]']
[MACRO RESTRICT '[HEX S2B + 2C2H]']
[MACRO SIZE '[HEX S2B + 2C7H]']
[MACRO UNPROTECT '[HEX S2B + 31FH]']
[MACRO WRITE$PRINTER '[HEX S3B + 0EDH]']
[MACRO SELECT$DRIVE '[HEX S3B + 0F2H]']
[MACRO SET$DMA '[HEX S3B + 15FH]']
[MACRO SET$TRACK '[HEX S3B + 17CH]']
[MACRO SET$SECTOR '[HEX S3B + 186H]']
[MACRO READ$FLOPPY '[HEX S3B + 1A0H]']
[MACRO WRITE$FLOPPY '[HEX S3B + 1B4H]']
[MACRO MTS$MSG '[HEX TB + 837H]']
[MACRO TERMINAL$STATUS '[HEX TB + 8D2H]']
[MACRO READ$TERMINAL '[HEX TB + 8DCH]']
[MACRO WRITE$TERMINAL '[HEX TB + 93CH]']

/*************** GENERAL PURPOSE MACROS ***************/

[INT TOP]  [TOP:=20]
[MACRO INPUT$WAITING '0FFH']

/******************** DECLARATIONS ********************/

DECLARE (MTS$EXTERNAL, MTS$INTERNAL, EXIT,
```

```
    MTS,TERM$BLOCK) LABEL;
DECLARE SAVE DATA (0,0,0);

/******************** ENTRY POINT ********************/

INTERNAL$MTS:  /* INTERNAL ENTRY POINT INTO SERVICE */
               /* MODULE, I.E. ENTRY POINT FROM     */
               /* OTHER MTS ROUTINES                */
    SAVE(2)=(A=LOCK);  /* SAVE LOCK VALUE */
    PARM=(HL=DE);  /* SAVE PARM LIST ADDRESS */
    SAVE=(HL=0+SP);   /* SAVE USER SP */
    SP=(HL=.SVC$STACK([TOP]));
    IF (A=<<C) !CY GOTO MTS;
    A=(A=!C)+1;  /* CONVERT FID TO POSITIVE NR */
    IF (A::3) !CY THEN
        /* INVALID COMMAND - ERROR 1 */
        DO; ERROR=(A=1); GOTO EXIT; END;
    H=0; L=A;
    DO CASE HL;
        /*  0 */  DO; NOP; END;
        /* -1 */  DO; CALL [READ$TERMINAL];
                      ERROR=A;
                  END;
        /* -2 */  CALL [BUMP];
    END; /* CASE */
    GOTO EXIT;

100H:  /* ADJUST EXTERNAL ENTRY POINT LOCATION */
EXTERNAL$MTS:  /* EXTERNAL ENTRY POINT INTO SERVICE */
               /* MODULE, I.E. ENTRY POINT FROM     */
               /* USER PROGRAMS                     */
    DISABLE;
    SAVE(2)=(A=LOCK  0FEH);  /* SAVE LOCK VALUE */
    LOCK=(A=A \ 01);  /* LOCK OUT SWAPPING */
    ENABLE;
    PARM=(HL=DE);  /* SAVE PARM LIST ADDRESS */
    SAVE=(HL=0+SP);   /* SAVE USER SP */
    SP=(HL=.SVC$STACK([TOP]));

MTS:  /* SYSTEM AND SERVICE ROUTINES */
    IF (A=C; A::18) !CY THEN
        /* INVALID COMMAND - ERROR 1 */
        DO; ERROR=(A=1); GOTO EXIT; END;
    ERROR=(A=0);  /* INITIALIZE RETURNED ERROR CODE */
    H=0; L=C;
    DO CASE HL;
        /******** SYSTEM CALLS ****************/
        /*  0 */  CALL [ATTACH];
        /*  1 */  CALL [MTS$MSG];
        /*  2 */  CALL [LOGIN];
        /*  3 */  CALL [PROTECT];
        /*  4 */  CALL [QUIT];
        /*  5 */  CALL [RESTRICT];
        /*  6 */  CALL [SIZE];
        /*  7 */  CALL [UNPROTECT];
        /******** SERVICE CALLS ***************/
        /*  8 */  DO; CALL [TERMINAL$STATUS];
                      ERROR=A;
                  END;
        /*  9 */  DO;  CALL [TERMINAL$STATUS];
                    IF (A::[INPUTS$WAITING]) ZERO THEN
                        DO; CALL [READ$TERMINAL];
                            ERROR=A;
                        END
                    ELSE GOTO TERM$BLOCK;
                  END;
        /* 10 */  CALL [WRITE$TERMINAL];
        /* 11 */  CALL [WRITE$PRINTER];
        /* 12 */  CALL [SELECT$DRIVE];
        /* 13 */  CALL [SET$DMA];
        /* 14 */  CALL [SET$TRACK];
        /* 15 */  CALL [SET$SECTOR];
```

```
        /* 16 */  CALL [READ$FLOPPY];
        /* 17 */  CALL [WRITE$FLOPPY];
    END; /* CASE */

EXIT:  /* COMMON EXIT POINT FOR INTERNAL AND EXTERNAL */
    SP=(HL=SAVE);  /* RESTORE USER SP */
    DISABLE;
    LOCK=(A=SAVE(2));  /* RESTORE LOCK VALUE */
    A=ERROR;
    ENABLE;
    RETURN;
    /* END MTS */

TERM$BLOCK:
/*************************************************************/
/* THIS ROUTINE IS CALLED WHEN THE TASK CURRENTLY            */
/* ALLOCATED THE CPU IS BLOCKED FOR TERMINAL I/O. THE        */
/* ROUTINE STORES THE CURRENT MACHINE ENVIRONMENT IN         */
/* THE SWAP STACK AND TRANSFERS CONTROL TO THE               */
/* MONITOR FOR SELECTION OF THE NEXT READY TASK.             */
/* CALLED BY: MTS                                            */
/*************************************************************/
    /* SET BIT 5 IN TCT$STATUS */
    DE=.TCT$STATUS; A=TASK; CALL [INDEX];
    M(HL)=(A=M(HL) \ 20H);
    /* SAVE ENVIRONMENT */
    SWAP$STACK=(HL=SAVE);  /* ONLY USER SP NEEDED */
    GOTO [MONITOR];
    /* END TERM$BLOCK */

EOF
/*****************************************************************/
/********************* SYSTEM CALLS **********************/
/*****************************************************************/


/************** INTERMODULE LINKAGE MACROS ************/

[INT TB MB M2B]  [M2B:=0600H] [TB:=1000H] [MB:=0300H]
[MACRO MONITOR '[HEX MB + 8FH]']
[MACRO INDEX '[HEX M2B + 3H]']
[MACRO INDEX2 '[HEX M2B + 0DH]']
[MACRO INDEX4 '[HEX M2B + 18H]']
[MACRO INDEX8 '[HEX M2B + 24H]']
[MACRO GET '[HEX M2B + 38H]']
[MACRO MOVBUF '[HEX M2B + 41H]']
[MACRO MINI$DISK '[HEX M2B + 54H]']
[MACRO SIZE$MSG '[HEX TB + 864H]']
[MACRO STATUS$MSG '[HEX TB + 88CH]']
[MACRO CLEAR$STATUS$LINE '[HEX TB + 827H]']
[MACRO MTS$MSG '[HEX TB + 837H]']

/***************** GENERAL PURPOSE MACROS ***************/

[MACRO WRITE '2']
[MACRO HARDWARE$ERROR '8']

/********************* DECLARATIONS *********************/

DECLARE PLIST DATA (1,0,0FFH,0FFH,0FFH,0FFH);

/******************* UTILITY PROCEDURES *****************/

VAL$DRIVE:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE VALIDATES THE DRIVE NUMBER INPUT TO        */
/* ANY SYSTEM CALL WHICH REQUIRES THAT PARAMETER.            */
/* INPUT: A - DRIVE NUMBER TO BE VALIDATED                   */
/* OUTPUT: DRIVE - NUMBER OF FREE DRIVE FOUND                */
/*         ERROR - ERROR CODE                                */
/* CALLED BY: ATTACH                                         */
/*************************************************************/
```

```
    IF (A::0FFH) ZERO THEN  /* NO DRIVE SPECIFIED */
        DO;  /* SCAN DRIVE MAP FOR FREE DRIVE */
            DECLARE L1 LABEL;
            DE=.TCTSDM; A=TASK; CALL [INDEX8];
            B=8;
            REPEAT;
                IF (A=<M(HL)) !CY GOTO L1;
                HL=HL+1;
            UNTIL (B=B-1) ZERO;
            /* NO DRIVE AVAILABLE - ERROR 10 */
            ERROR=(A=10); RETURN;
        L1: DRIVE=(A=8-B);  /* FREE DRIVE FOUND */
        END
    ELSE IF (A::8) !CY THEN
            /* DRIVE NR > 7 - ERROR 6 */
            DO; ERROR=(A=6); RETURN; END;
    END VAL$DRIVE;

VAL$DISK:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE VALIDATES THE DISK NUMBER INPUT TO  */
/* ANY SYSTEM CALL WHICH REQUIRES THAT PARAMETER.     */
/* INPUT: A - DISK NUMBER TO BE VALIDATED             */
/* OUTPUT: DISK - NUMBER OF FREE DISK FOUND           */
/*         ERROR - ERROR CODE                         */
/* CALLED BY: ATTACH                                  */
/*************************************************************/
    IF(A::0FFH) ZERO THEN  /* NO DISK SPECIFIED */
        DO;  /* SCAN DISK MAP FOR FREE DISK */
            DECLARE L2 LABEL;
            HL=.DMT$FLAG; B=32;
            REPEAT;
                IF (A=>M(HL)) CY  /* DISK AVAILABLE */
                  (A=>A) !CY      /* NOT IN USE     */
                  (A=>A) !CY      /* NOT PROTECTED  */
                  (A=>A) !CY      /* NOT RESTRICTED */
                  THEN GOTO L2;
                HL=HL+1;
            UNTIL (B=B-1) ZERO;
            /* NO DISK AVAILABLE - ERROR 2 */
            ERROR=(A=2); RETURN;
        L2: DISK=(A=32-B);  /* FREE DISK FOUND */
        END
    ELSE IF (A::32) !CY THEN
            /* DISK NR > 31 - ERROR 4 */
            DO; ERROR=(A=4); RETURN; END
        ELSE
            DO; /* SEE IF SPECIFIED DISK AVAILABLE */
            DE=.DMT$FLAG; A=DISK; CALL [INDEX];
            IF (A=>M(HL)) !CY THEN
                /* DISK NOT AVAILABLE - ERROR 2 */
                DO; ERROR=(A=2); RETURN; END;
            IF (A=>A) CY THEN
                /* DISK IN USE - ERROR 3 */
                DO; ERROR=(A=3); RETURN; END;
            END;
    END VAL$DISK;

VAL$KEY:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE COMPARES THE KEY INPUT AS A SYSTEM  */
/* CALL PARAMETER WITH THAT ASSOCIATED WITH A SPECI-  */
/* FIED VIRTUAL DISK FILE.                            */
/* INPUT: PARM - VARIABLE HOLDING ADDRESS OF PARM KEY */
/*        DISK - VIRTUAL DISK FILE NUMBER             */
/* OUTPUT: ERROR - ERROR CODE                         */
/* CALLED BY: ATTACH                                  */
/*************************************************************/
    DE=.DMT$KEY; A=DISK; CALL [INDEX4];
    DE=HL; HL=PARM;
    DO B=0 BY B=B+1 WHILE (A=4; A::B) !ZERO;
        IF (A=M(DE); A::M(HL)) ZERO
```

```
          \ (IF (A::20H) ZERO  (A=M(HL)-0FFH) ZERO
              THEN CY=1 ELSE CY=0) CY
          THEN  /* CHAR MATCH */
              DO;
              DE=DE+1; HL=HL+1;
              END
          ELSE  /* KEY ERROR - ERROR 5 */
              DO; ERROR=(A=5); RETURN; END;
      END; /* WHILE */
      /* KEYS MATCH */
      END VALSKEY;

CLEAR$FLAG:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE RESETS THE IN USE BIT (BIT 1) IN           */
/* THE DMT$FLAG FOR A SPECIFIED VIRTUAL DISK.                */
/* INPUT: B - DISK NUMBER                                    */
/* CALLED BY: ATTACH, LOGIN, CLEAR$DM                        */
/*************************************************************/
      DE=.DMT$FLAG; A=B  1FH; CALL [INDEX];
      M(HL)=(A=M(HL)  0FDH);
      END CLEAR$FLAG;

CLEAR$DM:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE RESETS ALL ENTRIES IN THE TCT$DM           */
/* ASSOCIATED WITH THE CURRENT VALUE OF TASK.                */
/* CALLED BY: LOGIN, QUIT                                    */
/*************************************************************/
      DE=.TCT$DM; A=TASK; CALL [INDEX8];
      M(HL)=0C0H; C=7;
      REPEAT;
          HL=HL+1;
          B=M(HL); M(HL)=0;
          IF (A=<B) CY  (A=<A) !CY THEN
              CALL CLEAR$FLAG;
      UNTIL (C=C-1) ZERO;
      END CLEAR$DM;

WRITE$BUF:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE CHECKS THE MODIFICATION BIT IN             */
/* VDC$DRIVE TO DETERMINE IF THE CONTENTS OF MDBUF           */
/* HAVE BEEN ALTERED. IF SO, THE BUFFER IS WRITTEN TO        */
/* THE MINI-DISK AND THE MOD BIT IS RESET.                   */
/* CALLED BY: QUIT, MAP, LOGIN                               */
/*************************************************************/
      IF (A=<VDC$DRIVE) !CY RETURN;
      BC=(HL=MDSAD); DE=.MDBUF;
      L=[WRITE]; CALL [MINI$DISK];
      IF (A::0) !ZERO THEN
          /* HARDWARE ERROR - ERROR 8 */
          DO; ERROR=(A=8); RETURN; END;
      VDC$DRIVE=(A=VDC$DRIVE  7FH);
      END WRITE$BUF;

/****************** SYSTEM ROUTINES ******************/

ATTACH:  PROCEDURE;
/*************************************************************/
/* SIMULATE THE PHYSICAL OPERATION OF LOADING DISK           */
/* <DISK NR> INTO DRIVE <DRIVE LTR>.                         */
/* ARGUMENTS:                                                */
/*  (1) FID = 0                                              */
/*  (2) PARM = BASE ADDRESS OF PARAMETER LIST                */
/*        BYTE 0: DRIVE NUMBER (0-7)                         */
/*        BYTE 1: DISK NUMBER (0-31)                         */
/*        BYTES 2-5: PROTECTION KEY (0-4 CHARS)              */
/* VALUE: ERROR CODE                                         */
/*************************************************************/
      BC=(HL=PARM); DRIVE=(A=M(BC));
      /* VALIDATE DRIVE NR */
```

```
    CALL VAL$DRIVE; IF (A=ERROR \ A) !ZERO RETURN;
    HL=PARM+1; DISK=(A=M(HL));
    /* VALIDATE DISK NR */
    CALL VAL$DISK; IF (A=ERROR \ A) !ZERO RETURN;
    DE=.DMT$FLAG; A=DISK; CALL [INDEX];
    IF (A=M(HL)  04H) !ZERO THEN  /* DISK PROTECTED */
        DO;  /* VALIDATE KEY */
        PARM=(HL=PARM+1,+1); CALL VAL$KEY;
        IF (A=ERROR \ A) !ZERO THEN
            DO;  /* MAY WANT READ ONLY */
            DE=.DMT$FLAG; A=DISK; CALL [INDEX]; BC=HL;
            IF (A=M(HL)  08H) ZERO
                        /* DISK NOT RESTRICTED */
             \ (HL=PARM; A=M(HL); A::0FFH) !ZERO
                        /* KEY PARM NON-BLANK */
              THEN RETURN   /* ERROR VALID */
            ELSE
                DO;  /* SET UP READ ONLY */
                M(BC)=(A=M(BC) \ 02H);
                        /* SET DMT$FLAG BIT 1 */
                DISK=(A=DISK \ 40H);
                        /* SET TCT$DM BIT 6 */
                ERROR=(A=0);
                        /* CLEAR ERROR */
                END;
            END;
        END;
    /* MODIFY DMT$FLAG */
    DE=.DMT$FLAG; A=DISK  1FH; CALL [INDEX];
    M(HL)=(A=M(HL) \\ 02H);  /* SET DMT$FLAG BIT 1 */
    /* MODIFY TCT$DM */
    DE=.TCT$DM; A=TASK; CALL [INDEX3];
    DE=HL; A=DRIVE; CALL [INDEX];
    B=M(HL);
    M(HL)=(A=DISK \ 80H);  /* SET TCT$DM BIT 7 */
    /* RESET OLD DISK'S IN USE BIT */
    IF (A=B  40H) ZERO CALL CLEAR$FLAG;
    /* DISPLAY STATUS MSG */
    B=(A=DRIVE); C=(A=DISK  1FH);
    IF (A=DISK 40H) !ZERO THEN A=72H ELSE A=' ';
    CALL [STATUS$MSG];
    END ATTACH;

LOGIN:  PROCEDURE;
/*****************************************************/
/* THIS SYSTEM CALL NOTIFIES MTS THAT THE REQUESTING  */
/* TERMINAL IS NOW ACTIVE, AND SIMULATES THE PHYSICAL */
/* COLD-START BOOTSTRAP OPERATION. THE BOOTSTRAP LOAD */
/* TAKES PLACE FROM VIRTUAL DRIVE A. THE VIRTUAL DISK */
/* ATTACHED TO THIS DRIVE MAY BE SPECIFIED IN THE     */
/* PARAMETER LIST, OR DISK0 WILL BE ASSUMMED AS THE   */
/* DEFAULT.                                           */
/* ARGUMENTS:                                         */
/*  (1) FID = 2                                       */
/*  (2) PARM = BASE ADDRESS OF PARAMETER LIST         */
/*       BYTE 0: DISK NUMBER (0-31)                   */
/*       BYTES 1-4: PROTECTION KEY (0-4 CHARS)        */
/* VALUE: ERROR CODE                                  */
/*****************************************************/
    /* WRITE MINI-DISK BUFFER IF NECESSARY */
    CALL WRITE$BUF;
    IF (A=ERROR \ A) !ZERO RETURN;
    /* RE-INITIALIZE TCT */
    CALL CLEAR$DM;
    DE=.TCT$SIZE; A=TASK; CALL [INDEX];
    M(HL)=32;  /* SIZE = 16K */
    HL=BC+(DE=.TCT$STATUS);
    M(HL)=1;  /* SET BIT 0 */
    /* PROCESS DISK PARAMETER */
    IF (HL=PARM; A=M(HL); A::0FFH) !ZERO THEN
        DO;  /* DISK SPECIFIED */
        BC=5; DE=HL; HL=.PLIST(1);
```

```
        CALL [MOVBUF]; PARM=(HL=.PLIST);
        CALL ATTACH;
        IF (A=ERROR \ A) !ZERO RETURN;
        END
    ELSE   /* DISPLAY DEFAULT STATUS MSG */
        DO;
        A=72H; BC=0;
        CALL [STATUS$MSG];
        END;
    /* DISPLAY SIZE MSG */
    A=16; CALL [SIZE$MSG];
    END LOGIN;

PROTECT:  PROCEDURE;
/*************************************************************/
/* THIS SYSTEM CALL ADDS THE READ/WRITE PROTECTION   */
/* ATTRIBUTE TO A SPECIFIED VIRTUAL DISK FILE.       */
/* ARGUMENTS:                                        */
/*  (1) FID = 3                                      */
/*  (2) PARM = BASE ADDRESS OF PARAMETER LIST        */
/*       BYTE 0: DISK NUMBER (0-31)                  */
/*       BYTES 1-4: PROTECTION KEY (0-4 CHARS)       */
/* VALUE: ERROR CODE                                 */
/*************************************************************/
    NOP;
    END PROTECT;

QUIT:  PROCEDURE;
/*************************************************************/
/* THIS SYSTEM CALL NOTIFIES MTS THAT THE REQUESTING */
/* TERMINAL IS NO LONGER ACTIVE.                     */
/* ARGUMENTS:                                        */
/*  (1) FID = 4                                      */
/*  (2) PARM = NONE                                  */
/* VALUE: NONE                                       */
/*************************************************************/
    /* WRITE MINI-DISK BUFFER IF NECESSARY */
    CALL WRITE$BUF;
    IF (A=ERROR \ A) !ZERO THEN
        DO; E=[HARDWARE$ERROR]; CALL [MTS$MSG]; END;
    /* CLEAR STATUS LINE */
    A=TASK; CALL [CLEAR$STATUS$LINE];
    /* CLEAR TCT */
    CALL CLEAR$DM;
    DE=.TCT$SIZE; A=TASK; CALL [INDEX];
    M(HL)=32;  /* SIZE = 16K */
    HL=BC+(DE=.TCT$STATUS);
    M(HL)=0;  /* RESET STATUS BYTE */
    GOTO [MONITOR];
    END QUIT;

BUMP:  PROCEDURE;
    CALL WRITE$BUF;
    IF (A=ERROR \ A) !ZERO THEN
        DO; E=[HARDWARE$ERROR]; CALL [MTS$MSG]; END;
    A=TASK; CALL [CLEAR$STATUS$LINE];
    CALL CLEAR$DM;
    DE=.TCT$SIZE; A=TASK; CALL [INDEX];
    M(HL)=32;
    HL=BC+(DE=.TCT$STATUS);
    M(HL)=0;
    END BUMP;

RESTRICT:  PROCEDURE;
/*************************************************************/
/* THIS SYSTEM CALL ADDS THE READ RESTRICTION ATTRI- */
/* BUTE TO A SPECIFIED PROTECTED VIRTUAL DISK FILE.  */
/* ARGUMENTS:                                        */
/*  (1) FID = 5                                      */
/*  (2) PARM = BASE ADDRESS OF PARAMETER LIST        */
/*       BYTE 0: DISK NUMBER (0-31)                  */
/*       BYTES 1-4: PROTECTION KEY (0-4 CHARS)       */
```

```
/* VALUE: ERROR CODE                                              */
/*****************************************************************/
    NOP;
    END RESTRICT;

SIZE:  PROCEDURE;
/*****************************************************************/
/* THIS SYSTEM CALL SETS THE SIZE OF THE USER'S SWAP             */
/* IMAGE.                                                        */
/* ARGUMENTS:                                                    */
/*  (1) FID = 6                                                  */
/*  (2) PARM = REQUESTED SIZE IN KILOBYTES                       */
/* VALUE: ERROR CODE                                             */
/*****************************************************************/
    /* COMPARE REQUESTED SIZE WITH MAX SIZE */
    IF (A=PARM(0); A::49) !CY THEN
        /* OUT OF BOUNDS - ERROR 7 */
        DO; ERROR=(A=7); RETURN; END;
    /* COMPARE REQUESTED SIZE WITH SWAP FILE SIZE */
    DE=.TCT$BOE; A=TASK; CALL [INDEX2];
    CALL [GET]; B=0; C=(A=<<PARM(0));
    HL=BC+DE; STACK=HL;  /* SAVE SUM */
    DE=.TCT$EOE; A=TASK; CALL [INDEX2];
    CALL [GET]; BC=BC+1;
    DE=STACK;  /* RESTORE SUM */
    A=C-E; A=B--D;
    IF MINUS THEN
        /* OUT OF BOUNDS - ERROR 7 */
        DO; ERROR=(A=7); RETURN; END;
    /* DISPLAY SIZE MSG */
    A=PARM(0); CALL [SIZE$MSG];
    /* UPDATE TCT */
    DE=.TCT$SIZE; A=TASK; CALL [INDEX];
    M(HL)=(A=<<PARM(0));
    END SIZE;

UNPROTECT:  PROCEDURE;
/*****************************************************************/
/* THIS SYSTEM CALL DELETES THE READ RESTRICTION AND             */
/* READ/WRITE PROTECTION ATTRIBUTES FROM A SPECIFIED             */
/* VIRTUAL DISK FILE.                                            */
/* ARGUMENTS:                                                    */
/*  (1) FID = 7                                                  */
/*  (2) PARM = BASE ADDRESS OF PARAMETER LIST                    */
/*       BYTE 0: DISK NUMBER (0-31)                              */
/*       BYTES 1-4: PROTECTION KEY (0-4 CHARS)                   */
/* VALUE: ERROR CODE                                             */
/*****************************************************************/
    NOP;
    END UNPROTECT;


EOF
/*****************************************************************/
/********************* SERVICE CALLS *****************************/
/*****************************************************************/


/************** INTERMODULE LINKAGE MACROS ************/

[INT M2B S2B]  [M2B:=0600H]  [S2B:=2200H]
[MACRO INDEX '[HEX M2B + 3H]']
[MACRO INDEX2 '[HEX M2B + 0DH]']
[MACRO INDEX8 '[HEX M2B + 24H]']
[MACRO GET '[HEX M2B + 38H]']
[MACRO MOVBUF '[HEX M2B + 41H]']
[MACRO MINI$DISK '[HEX M2B + 54H]']

/***************** GENERAL PURPOSE MACROS ***************/

[MACRO READ '1']
[MACRO WRITE '2']
```

```
/****************** UTILITY PROCEDURES ******************/

READSBUF:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE READS A SPECIFIED MINI-DISK SECTOR  */
/* INTO MDBUF AND UPDATES MDSAD.                      */
/* INPUT: BC - MINI-DISK SECTOR NUMBER                */
/* OUTPUT: ERROR - ERROR CODE                         */
/* CALLED BY: MAP                                     */
/*************************************************************/
    DE=.MDBUF; L=[READ]; CALL [MINISDISK];
    IF (A::0) !ZERO THEN
        /* HARDWARE ERROR - ERROR 8 */
        DO; ERROR=(A=8); RETURN; END;
    MDSAD=(HL=BC);
    END READSBUF;

WRITESBUF:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE CHECKS THE MODIFICATION BIT IN      */
/* VDCSDRIVE TO DETERMINE IF THE CONTENTS OF MDBUF    */
/* HAVE BEEN ALTERED. IF SO, THE BUFFER IS WRITTEN TO */
/* THE MINI-DISK AND THE MOD BIT IS RESET.            */
/* CALLED BY: QUIT, MAP, LOGIN                        */
/*************************************************************/
    IF (A=<VDCSDRIVE) !CY RETURN;
    BC=(HL=MDSAD); DE=.MDBUF;
    L=[WRITE]; CALL [MINISDISK];
    IF (A::0) !ZERO THEN
        /* HARDWARE ERROR - ERROR 8 */
        DO; ERROR=(A=8); RETURN; END;
    VDCSDRIVE=(A=VDCSDRIVE  7FH);
    END WRITESBUF;

MAP:  PROCEDURE;
/*************************************************************/
/* THIS PROCEDURE CALCULATES THE RELATIVE OFFSET OF A */
/* SPECIFIED VIRTUAL FLOPPY DISK SECTOR IN THE MINI-  */
/* DISK FILE AND THE BASE ADDRESS OF THAT SECTOR IN   */
/* THE MINI-DISK BUFFER AFTER THE FILE IS READ. THEN  */
/* IT CALCULATES THE ACTUAL MINI-DISK SECTOR NUMBER   */
/* CONTAINING THE ADDRESSED FLOPPY DISK SECTOR, AND   */
/* COMPARES IT WITH THE CURRENT CONTENTS OF MDBUF. IF */
/* THE TWO ARE NOT EQUAL, THE OLD BUFFER IS WRITTEN   */
/* TO THE MINI-DISK AND THE NEWLY CALCULATED SECTOR   */
/* NUMBER READ IN TO REFILL THE BUFFER. A COMPARISON  */
/* IS ALSO MADE BETWEEN THE CALCULATED MINI-DISK      */
/* SECTOR NUMBER AND THE FILE'S EOE VALUE TO ENSURE   */
/* THAT THE SPECIFIED VIRTUAL DISK SECTOR ADDRESS IS  */
/* WITHIN THE BOUNDS OF THE FILE.                     */
/* INPUT:  VDCSTRACK - FLOPPY DISK TRACK NUMBER       */
/*         VDCSSECTOR - FLOPPY DISK SECTOR NUMBER     */
/* OUTPUT:  BC = 128 = FLOPPY DISK SECTOR SIZE        */
/*          HL = BASE ADDRESS OF FLOPPY DISK SECTOR   */
/*               IN BUFFER                            */
/* CALLED BY:  READSFLOPPY, WRITESFLOPPY              */
/*************************************************************/
    DECLARE BUFAD DATA (0,0);
    DECLARE SECNR DATA (0,0);
    /* MULTIPLY TRACK NUMBER BY 26 */
    D=(A=VDCSTRACK);
    E=(A=VDCSSECTOR);
    B=0; C=26; H=8;
    C=(A=>C);
    REPEAT;
        IF CY THEN A=B+D;
        CY=0; B=(A=>A);
        C=(A=>C);
    UNTIL (H=H-1) ZERO;     /* BC = 26 * TRACK */
    /* ADD SECTOR NUMBER TO 26 * TRACK */
    C=(A=C+E); B=(A=B++0);
```

```
                /* BC = 26 * TRACK + SECTOR */
    /* DIVIDE 26 * TRACK + SECTOR BY 4 */
    DE=0;
    CY=0; B=(A=>B); C=(A=>C); E=(A=>E);
    CY=0; B=(A=>B); C=(A=>C); D=(A=<D);
                /* BC = BC / 4 */
                /* DE = RELATIVE BUFFER ADDRESS */
    BUFAD=(HL=DE);  /* SAVE RELATIVE BUFFER ADDRESS */
    /* CALCULATE AND SAVE NEW MINI-DISK SECTOR NR */
    SECNR=(HL=VDC$BOE + BC);
    /* COMPARE NEW SECTOR NR WITH VDC$EOE */
    A=VDC$EOE(1)-L; A=VDC$EOE(0)--H;
    IF MINUS THEN
        /* OUT OF BOUNDS - ERROR 7 */
        DO; ERROR=(A=7); RETURN; END;
    /* COMPARE NEW SECTOR NR WITH MDSAD */
    IF (A=MDSAD(0); A::H) !ZERO THEN
        CY=0
    ELSE IF (A=MDSAD(1); A::L) !ZERO THEN
            CY=0
        ELSE CY=1;
    /* WRITE OLD SECTOR AND READ NEW IF NECESSARY */
    IF !CY THEN
        DO;  /* NOT EQUAL */
        CALL WRITE$BUF;
        IF (A=ERROR \ A) !ZERO RETURN;
        BC=(HL=SECNR); CALL READ$BUF;
        IF (A=ERROR \ A) !ZERO RETURN;
        END;
    /* SET UP REGISTERS FOR RETURN */
    BC=128; DE=(HL=BUFAD);
    END MAP;

/******************* SERVICE ROUTINES ******************/

WRITE$PRINTER:  PROCEDURE;
/*********************************************************/
/* THIS SERVICE CALL IS CALLED TO WRITE A SINGLE       */
/* CHARACTER TO THE SERIAL PRINTER.                    */
/* ARGUMENTS:                                          */
/*  (1) FID = 11                                       */
/*  (2) PARM = ASCII CHARACTER                         */
/* VALUE: ERROR CODE                                   */
/*********************************************************/
    NOP;
    END WRITE$PRINTER;

SELECT$DRIVE:  PROCEDURE;
/*********************************************************/
/* THIS SERVICE CALL SELECTS THE VIRTUAL FLOPPY DISK   */
/* DRIVE TO BE USED FOR SUBSEQUENT FLOPPY DISK         */
/* ACCESSES.                                           */
/* ARGUMENTS:                                          */
/*  (1) FID = 12                                       */
/*  (2) PARM = DRIVE NUMBER (0-7)                      */
/* VALUE: ERROR CODE                                   */
/*********************************************************/
    DRIVE=(A=PARM(1));
    /* VALIDATE DRIVE NUMBER */
    IF (A::8) !CY THEN
        /* DRIVE NR > 7 - ERROR 6 */
        DO; ERROR=(A=6); RETURN; END;
    /* VALIDATE THAT DRIVE IN USE */
    DE=.TCT$DM; A=TASK; CALL [INDEX$];
    DE=HL; A=DRIVE; CALL [INDEX];
    IF (A=<M(HL)) !CY THEN
        /* DRIVE NOT AVAIL - ERROR 10 */
        DO; ERROR=(A=10); RETURN; END;
    /* UPDATE VDC BLOCK */
    DISK=(A=M(HL)  1FH);
    IF (A=M(HL)  40H) !ZERO THEN  /* READ ONLY */
        DRIVE=(A=DRIVE \ 40H);  /* SET BIT 6 */
```

```
    DE=.DMT$BOE; A=DISK; CALL [INDEX2];
    CALL [GET]; VDC$BOE=(HL=BC);
    DE=.DMT$EOE; A=DISK; CALL [INDEX2];
    CALL [GET]; VDC$EOE=(HL=BC);
    VDC$DRIVE=(A=DRIVE);
    END SELECT$DRIVE;

SET$DMA:  PROCEDURE;
/*************************************************************/
/* THIS SERVICE CALL SETS THE ADDRESS OF THE 128 BYTE */
/* DMA BUFFER TO BE USED IN SUBSEQUENT VIRTUAL FLOPPY */
/* DISK ACCESSES.                                             */
/* ARGUMENTS:                                                 */
/*  (1) FID = 13                                              */
/*  (2) PARM = DMA ADDRESS                                    */
/* VALUE: ERROR CODE                                          */
/*************************************************************/
    A=PARM(1)-0; A=PARM(0)--40H;
    IF MINUS THEN
        /* OUT OF BOUNDS - ERROR 7 */
        DO; ERROR=(A=7); RETURN; END;
    VDC$DMA=(HL=PARM);
    END SET$DMA;

SET$TRACK:  PROCEDURE;
/*************************************************************/
/* THIS SERVICE CALL SETS THE TRACK NUMBER TO BE USED */
/* IN SUBSEQUENT VIRTUAL FLOPPY DISK ACCESSES.                */
/* ARGUMENTS:                                                 */
/*  (1) FID = 14                                              */
/*  (2) PARM = TRACK NUMBER                                   */
/* VALUE: NONE                                                */
/*************************************************************/
    VDC$TRACK=(A=PARM(1));
    END SET$TRACK;

SET$SECTOR:  PROCEDURE;
/*************************************************************/
/* THIS SERVICE CALL SETS THE SECTOR NUMBER TO BE     */
/* USED IN SUBSEQUENT VIRTUAL FLOPPY DISK ACCESSES.   */
/* ARGUMENTS:                                                 */
/*  (1) FID = 15                                              */
/*  (2) PARM = SECTOR NUMBER (1-26)                           */
/* VALUE: ERROR CODE                                          */
/*************************************************************/
    A=PARM(1);
    IF (A::27) !CY  \ (A::0) ZERO THEN
        /* OUT OF BOUNDS - ERROR 7 */
        DO; ERROR=(A=7); RETURN; END;
    VDC$SECTOR=A;
    END SET$SECTOR;

READ$FLOPPY:  PROCEDURE;
/*************************************************************/
/* THIS SERICE CALL SIMULATES READING FROM A FLOPPY   */
/* DISK. THE VIRTUAL DISK SECTOR AND TRACK SPECIFIED  */
/* IN THE VDC BLOCK IS READ FROM THE SPECIFIED        */
/* VIRTUAL DRIVE INTO A 128 BYTE BUFFER IN THE USER'S */
/* SWAP AREA.                                                 */
/* ARGUMENTS:                                                 */
/*  (1) FID =16                                               */
/*  (2) PARM = NONE                                           */
/*      DRIVE, SECTOR, TRACK, AND DMA ADDRESS MUST    */
/*      HAVE BEEN PREVIOUSLY SET BY CALLS TO THE      */
/*      APPROPRIATE PROCEDURES.                       */
/* VALUE: ERROR CODE                                          */
/*************************************************************/
    CALL MAP;
    IF (A=ERROR \ A) !ZERO RETURN;
    DE=HL; HL=VDC$DMA; CALL [MOVBUF];
    END READ$FLOPPY;
```

```
WRITESFLOPPY:  PROCEDURE;
/*****************************************************/
/* THIS SERVICE CALL SIMULATES WRITING TO A FLOPPY   */
/* DISK. A 128 BYTE BUFFER IN THE USER SWAP AREA IS  */
/* WRITTEN TO THE VIRTUAL TRACK, SECTOR, AND DRIVE   */
/* SPECIFIED IN THE VDC BLOCK.                       */
/* ARGUMENTS:                                        */
/*  (1) FID = 17                                     */
/*  (2) PARM = NONE                                  */
/*      DRIVE, SECTOR, TRACK, AND DMA ADDRESS MUST   */
/*      HAVE BEEN PREVIOUSLY SPECIFIED BY CALLS TO   */
/*      THE APPROPRIATE PROCEDURES.                  */
/* VALUE: ERROR CODE                                 */
/*****************************************************/
    CALL MAP;
    IF (A=ERROR \ A) !ZERO RETURN;
    STACK=HL; DE=(HL=VDCSDMA);
    HL=STACK; CALL [MOVBUF];
    VDCSDRIVE=(A=VDCSDRIVE \ 80H);  /* SET BIT 7 */
    END WRITESFLOPPY;

EOF
```

```
/*****************************************************/
/************  TERMINAL INTERFACE MODULE  ************/
/*****************************************************/
/*                                                   */
/*      TERMINAL INTERFACE MODULE PROVIDES THE MTS   */
/* INTERFACE WITH THE FOUR DISPLAY TERMINALS         */
/* ATTACHED TO THE SYCOR 440 SYSTEM.  THE MODULE IS  */
/* SEPARATED INTO FIVE BASIC SUBMODULES.  EACH       */
/* SUBMODULE CONTAINS THE MACROS NECESSARY FOR       */
/* LINKING WITHIN THE TERMINAL MODULE AND FOR        */
/* EXTERNAL MTS LINKAGE.                             */
/*                                                   */
/*  (1) DATA DECLARATIONS                            */
/*      THIS SUBMODULE PROVIDES ALL DECLARATIONS OF  */
/*      THE DATA STRUCTURES UTILIZED BY THE TERMINAL */
/*      MODULE.                                      */
/*  (2) UTILITY PROCEDURES                           */
/*      THIS SUBMODULE CONTAINS THE BASIC UTILITY    */
/*      PROCEDURES WHICH PROVIDE COMMON REGISTER     */
/*      MANIPULATION AND PROCESSING REQUIRED BY      */
/*      MANY PROCEDURES THROUGHOUT THE REMAINDER     */
/*      OF THE TERMINAL INTERFACE MODULE.            */
/*          * COMPARE$PTRS       * GET$INDEX         */
/*          * GET$VALUE          * STORE$VALUE       */
/*          * CONVERT$NUMBR$TO$ASCII                 */
/*          * MOVE$BYTES         * SWAP$CURSOR       */
/*  (3) TERMINAL INTERFACE PRIMITIVES                */
/*      THIS SUBMODULE CONTAINS THE PROCEDURES WHICH */
/*      PROVIDE THE BASIC TERMINAL INTERFACE         */
/*      FUNCTIONS.                                   */
/*          * BLANK$DISPLAY      * GET$DISPLAYS$ADDR */
/*          * CHECK$CURSOR       * GET$TERM$STATUS   */
/*          * SCROLL$DISPLAY     * SEND$BEEP         */
/*          * SEND$CLICK         * UPDATE$CURSOR     */
/*          * GET$STATUS$ADDR                        */
/*  (4) KEY PROCESSING PROCEDURES                    */
/*      THIS SUBMODULE CONTAINS THE PROCEDURES WHICH */
/*      PROVIDE THE BASIC TERMINAL INPUT PROCESSING. */
/*      THEY ARE CALLED TO PROCESS EACH KEY ENTERED  */
/*      AT A TERMINAL.  THEIR FUNCTIONS INCLUDE:     */
/*      CHECKING FOR AND CONVERTING LOWER TO UPPER   */
/*      CASE LETTERS IF REQUIRED; CHECKING FOR ANY   */
/*      KEY COMMANDS (WHICH INCLUDES ALL LINE        */
/*      EDITING); AND IF NOT A KEY COMMAND,          */
/*      DISPLAYING THE INPUT CHAR AT THE TERMINAL.   */
/*          * KEY$COMMAND    * TERM$INPUT$CNTL       */
/*  (5) TERMINAL INTERFACE SYSTEM FUNCTIONS          */
/*      THIS SUBMODULE CONTAINS THE PROCEDURES WHICH */
/*      PROVIDE THE PROCESSING REQUIRED TO INTERFACE */
/*      THE TERMINAL WITH THE REST OF THE MTS        */
/*      MODULES.  IT PROVIDES READING AND WRITING    */
/*      OF CHARACTERS FROM/TO THE TERMINALS;         */
/*      TERMINAL STATUS INFORMATION (E.G. WHETHER    */
/*      THERE'S INPUT AVAILABLE OR NOT); AND DISPLAY */
/*      OF STATUS AND MTS MESSAGES ON THE TERMINAL   */
/*      STATUS LINE.                                 */
/*          * BLINK$CURSORS      * CLEAR$STATUS$LINE */
/*          * TERMINAL$STATUS    * READ$TERMINAL     */
/*          * WRITE$TERMINAL     * STATUS$MSG        */
/*          * MTS$MSG            * SIZE$MSG          */
/*****************************************************/
```

```
/*************************************************/
/************  TERMINAL DISPLAY DESIGN  **********/
/*                                               */
/*                                               */
/*       GENERAL FORMAT OF EACH TERMINAL DISPLAY */
/*                                               */
/*                                               */
/*       ----------------------------------------*/
/*       |0            STATUS LINE           63|*/
/*       ----------------------------------------*/
/*       |////////////////////////////////////////|*/
/*       ----------------------------------------*/
/*       |0     D                              63|*/
/*       ----------------------------------------*/
/*       |64        I                         127|*/
/*       ----------------------------------------*/
/*       |128   B       S                     191|*/
/*       ----------------------------------------*/
/*       |192       U       P                 255|*/
/*       ----------------------------------------*/
/*       |256           F       L             319|*/
/*       ----------------------------------------*/
/*       |320               F       A         383|*/
/*       ----------------------------------------*/
/*       |384                   E       Y     447|*/
/*       ----------------------------------------*/
/*       |448                       R         511|*/
/*       ----------------------------------------*/
/*                                               */
/*                                               */
/*                 STATUS LINE                   */
/*       ----------------------------------------*/
/*       |0      VFD       39|40 MS 47|48 MSG 63|*/
/*       ----------------------------------------*/
/*       WHERE                                   */
/*      VFD - VIRTUAL FLOPPY DISK STATUS DISPLAY;*/
/*            CONTAINS THE INFORMATION ASSOCIATED*/
/*            WITH THE CURRENT TERMINAL USER'S   */
/*            VIRTUAL FLOPPY DISK DRIVE AND DISK */
/*            NUMBER ASSIGNMENTS. (SEE STATUS$MSG*/
/*            PROC FOR DETAILS)                  */
/*       MS - DISPLAY OF THE MEMORY SIZE THE USER*/
/*            HAS REQUESTED. (SEE SIZE$MSG PROC) */
/*      MSG - DISPLAY AREA FOR MTS MESSAGES WHICH*/
/*            ARE DISPLAYED IN RESPONSE TO A     */
/*            SYSTEM OR SERVICE CALL TO MTS.     */
/*            (SEE MTS$MSG PROC)                 */
/*                                               */
/*                                               */
/*                   DISPLAY BUFFER              */
/*                   --------------              */
/*                                               */
/* THE DISPLAY BUFFER IS VIEWED AS A SINGLE      */
/* BUFFER FROM 0 TO 512 BYTES IN LENGTH.  THERE ARE*/
/* FOUR DISPLAY BUFFER POINTERS WHICH PROVIDE THE*/
/* CONTROL OF INPUT FROM AND OUTPUT TO THE TERMINAL*/
/* DISPLAY.  THESE POINTERS ARE:  CURRENT$LINE;  */
/* CURSOR; NEXT$CHAR; AND END$IBUFF.             */
/* EACH POINTER UTILIZES TWO BYTES OF STORAGE TO */
/* ACCOMMODATE A RANGE IN VALUE FROM 0 TO 512.   */
/* IN ADDITION TO THESE POINTERS, THERE IS A     */
/* TERMINAL STATUS BYTE, CALLED TERM$STATUS,     */
/* ASSOCIATED WITH EACH TERMINAL.  IT IS SET TO  */
/* ONE OF THREE VALUES:  INPUT$WAITING;          */
/* MTS$CMD$READY; IBUFF$EMPTY.  THESE ARE THE    */
/* PRIMARY DATA STRUCTURES PROVIDING TERMINAL I/O */
/* CONTROL.                                      */
/*                                               */
/* THE PRIMARY SYCOR HARDWARE CHARACTERISTIC WHICH*/
/* AFFECTED THE MTS TERMINAL INTERFACE WAS THE   */
/* RELATIVELY SLOW MINI-DISK ACCESS TIMES.  THIS HAS*/
/* A MAJOR IMPACT WHEN TRYING TO DESIGN AN       */
```

```
/* INTERACTIVE TIMESHARED SYSTEM.  INORDER TO        */
/* PROVIDE REASONABLE INTERACTIVE RESPONSE TIMES     */
/* TO USER ACTIONS, THE TERMINAL INTERFACE           */
/* MODULE PROVIDES ALL LINE EDITING FEATURES FOR     */
/* THE USER PRIOR TO TRANSFERING ANY DATA TO THE     */
/* USER'S PROGRAM. (SEE KEYSCOMMAND PROC)            */
/* THE TERMINAL DISPLAY DESIGN UTILITZES TWO         */
/* SEPARATE BUFFERS TO PROVIDE THE USER WITH THE     */
/* CAPABILITY OF CONTINUING TO ENTER DATA PRIOR      */
/* TO THE USER'S PROGRAM BEING SWAPPED IN TO         */
/* PROCESS THE PREVIOUS INPUT LINE.                  */
/* THE FIRST BUFFER IS CALLED THE 'CURRENT LINE'     */
/* AND CONTAINS THE INPUT DATA WHICH IS CURRENTLY    */
/* BEING ENTERED BY THE USER.  THE CURRENT LINE CAN  */
/* RANGE FROM 0 TO 512 BYTES IN LENGTH.  THIS IS THE */
/* DATA THAT IS AFFECTED BY ANY LINE EDITING COMMAND */
/* ENTERED BY THE USER.                              */
/* THE SECOND BUFFER IS THE INPUT LINE OR BUFFER.    */
/* THE CURRENT LINE BECOMES THE INPUT LINE WHENEVER  */
/* A CARRIAGE RETURN OR ERROR RESET (MTS COMMAND     */
/* KEY) IS ENTERED.  THIS ACTION ALSO ESTABLISHES    */
/* A NEW CURRENT LINE.  THE INPUT LINE CONTAINS THE  */
/* THE DATA WHICH IS TRANSFERRED TO THE USER PROGRAM */
/* WHEN REQUESTED.  THUS THERE CAN BE AN INPUT LINE  */
/* AND A CURRENT LINE ESTABLISHED AT ONE TIME.       */
/* THE CURRENT CURSOR POSITION ALWAYS SPECIFIES      */
/* WHERE THE NEXT CHARACTER WILL BE ENTERED, ON      */
/* INPUT, AND WHERE THE NEXT CHARACTER WILL BE       */
/* DISPLAYED DURING OUTPUT.                          */
/*                                                   */
/* THE FOLLOWING IS AN EXAMPLE OF POINTER            */
/* MANIPULATION DURING INPUT:                        */
/*  INITIALIZATION OR CLEAR SCREEN CMD:              */
/*      ALL POINTERS ARE SET TO ZERO AND             */
/*      TERM$STATUS = INPUT BUFFER EMPTY.            */
/*  USER ENTERS DATA - "SAMPLE INPUT DATA":          */
/*      CURRENT$LINE POINTS TO THE STARTING POSITION */
/*      AND CURSOR ALWAYS POINTS TO THE NEXT         */
/*      POSITION TO FILL.  NOTE THAT AT THIS POINT   */
/*      ONLY CURSOR HAS BEEN MODIFIED. FOR THE       */
/*      INPUT DATA ABOVE IT WOULD BE POINTING TO     */
/*      DISPLAY BUFFER POSITION 17.                  */
/*  USER TERMINATES CURRENT LINE (I.E. ENTERS CR OR  */
/*  MTS$CMD):                                        */
/*      END$IBUFF IS SET TO CURRENT CURSOR POSITION. */
/*      CURSOR IS SET TO LEFT MOST POSITION OF NEXT  */
/*          LINE ON DISPLAY.                         */
/*      CURRENT$LINE IS SET TO NEW CURSOR POSITION.  */
/*      TERM$STATUS IS SET TO INPUT WAITING.         */
/*                                                   */
/* THE RESULTING POINTER POSITIONS ARE SHOWN FOR THE */
/* SAMPLE INPUT DATA AND CR CHARACTERS ENTERED.      */
/* (WHERE * = CURRENT CURSOR POSITION)               */
/*                                                   */
/*      NC                  EIB                      */
/*      ----------------------------------------     */
/*      |SAMPLE INPUT DATA                     |     */
/*      ----------------------------------------     */
/*      |*                                     |     */
/*      ----------------------------------------     */
/*      CL                                           */
/*                                                   */
/* THE SAMPLE INPUT DATA IS NOW AVAILABLE FOR THE    */
/* USER'S PROGRAM WHEN IT'S TIMESLICE COMES UP.      */
/* THE NEXT$CHAR (NC) POINTER SPECIFIES THE NEXT     */
/* CHARACTER TO BE READ AND RETURNED TO THE USER     */
/* PROGRAM.  WHEN NEXT$CHAR = END$IBUFF, A CARRIAGE  */
/* RETURN (CR) IS RETURNED TO THE CALLING USER       */
/* PROGRAM.                                          */
/* THERE ARE THREE OCCASIONS WHEN THE NEXT$CHAR      */
/* POINTER IS RESET EQUAL TO THE CURRENT$LINE        */
/* POINTER:                                          */
```

```
/*  (1) FOR A CLEAR SCREEN COMMAND.                          */
/*  (2) WHEN READ$TERMINAL PROC DETECTS THAT THE             */
/*      END OF THE INPUT BUFFER HAS BEEN REACHED.            */
/*  (3) WHEN WRITE$TERMINAL PROC OUTPUTS CHARACTERS          */
/*      TO THE TERMINAL FROM THE USER'S PROGRAM.             */
/*                                                           */
/* THE OUTPUT OF DATA FROM THE USER'S PROGRAM TO             */
/* THE TERMINAL RESULTS IN THE FOLLOWING:                    */
/*  (1) THE CHARACTER IS DISPLAYED AT THE CURRENT            */
/*      CURSOR POSITION.                                     */
/*  (2) THE CURSOR POSITION IS INCREMENTED.                  */
/*  (3) THE CURRENT$LINE AND NEXT$CHAR POINTERS ARE          */
/*      SET EQUAL TO THE NEW CURSOR POSITION.                */
/*  (4) THE TERMINAL STATUS IS SET TO EMPTY.                 */
/*                                                           */
/*                                                           */
/* ANOTHER DESIGN CONSIDERATION WAS THE REQUIREMENT          */
/* FOR THE TERMINAL MODULE TO PROVIDE A BLINKING             */
/* CURSOR DISPLAY AT EACH TERMINAL.  THIS REQUIRED           */
/* SPECIAL PROCESSING TO ENSURE THAT THE CURSOR              */
/* CHARACTER (05FH) DID NOT GET LOST DURING THE              */
/* CURSOR UPDATE AND MANIPULATION FUNCTIONS                  */
/* ACCOMPLISHED BY THE KEY PROCESSING AND SYSTEM             */
/* FUNCTION SUBMODULES. CHECK$CURSOR PROC(PRIMITIVE          */
/* SUBMODULE) PROVIDES THIS FUNCTION.                        */
/*                                                           */
/*                                                           */
/*************************************************************/
/*************************************************************/


/*************************************************************/
/******  TERMINAL INTERFACE DATA DECLARATIONS  ********/
/*************************************************************/


[MACRO IBUFF$EMPTY  '0'  ]
[MACRO CURSOR$CHAR  '5FH']

/*************************************************************/
/*********  TERMINAL INTERFACE DECLARATIONS  ********/

  /* ASCII - CONTAINS DATA FOR MATRIX CODE TO ASCII*/
  /*         CONVERSION.                           */

DECLARE ASCII DATA (1EH,1CH,1BH,5DH,5BH,29H,28H,7FH,
    26H,3DH,25H,24H,23H,40H,21H,2AH,0AH,0CH,0BH,0A0H,
    0DH,50H,4FH,15H,55H,39H,54H,52H,45H,57H,51H,49H,
    0DH,9,31H,9,22H,3AH,4CH,0DH,4AH,48H,47H,46H,44H,53H,
    41H,4BH,0FFH,30H,20H,0A3H,0A2H,3FH,3EH,3CH,4DH,4EH,
    42H,56H,43H,58H,5AH,0FFH,
    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,7FH,0FFH,0FFH,
    0FFH,0A4H,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0A0H,
    0DH,0FFH,0FFH,15H,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,
    0FFH,0FFH,0FFH,0FFH,0FFH,09,0FFH,0FFH,0FFH,0DH,
    0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,
    20H,0A3H,0A2H,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,0FFH,
    0A1H,0FFH,0FFH,0FFH,
    0FFH,0FFH,0FFH,5FH,0FFH,7DH,7BH,7FH,0FFH,7EH,5CH,
    0A4H,60H,5EH,7CH,0FFH,0FFH,0FFH,0FFH,0A0H,0DH,
    10H,0FH,15H,15H,19H,14H,12H,05,17H,11H,09,60H,
    5EH,7CH,09,0FFH,0FFH,0CH,0DH,0AH,08,07,06,04,
    13H,01,0BH,0FFH,7DH,20H,0A3H,0A2H,0FFH,0FFH,0FFH,
    0DH,0EH,02,16H,03,18H,1AH,0FFH,
    39H,38H,37H,2DH,2BH,30H,39H,7FH,37H,36H,35H,34H,
    33H,32H,31H,38H,36H,35H,34H,0A0H,0DH,70H,6FH,15H,
    75H,79H,74H,72H,65H,77H,71H,69H,33H,32H,31H,09,
    27H,3BH,6CH,0DH,6AH,68H,67H,66H,64H,73H,61H,6BH,
    0FFH,30H,20H,0A3H,0A2H,2FH,2EH,2CH,6DH,6EH,62H,
    76H,63H,78H,7AH,0FFH);

/*************************************************************/
```

```
/* STATUS$BASE  - START OF STATUS LINE AT EACH TERM */
/* DISPLAY$BASE - START OF DISPLAY LINES            */
/*************************************************/
DECLARE STATUS$BASE  DATA
                     (00H,07H,40H,09H,80H,0BH,0C0H,0DH);
DECLARE DISPLAY$BASE DATA
                     (40H,07H,80H,09H,0C0H,0BH,00H,0EH);

/*************************************************/
/* MTS$MESSAGE - DATA VECTOR CONTAINING ALL THE MTS */
/*               MESSAGES WHICH MAY BE DISPLAYED IN */
/*               THE MTS MSG FIELD OF THE STATUS   */
/*               LINE.                             */
/* SIZE$MESSAGE- DATA VECTOR CONTAINING THE TEXT   */
/*               PORTION OF THE SIZE MSG FIELD OF  */
/*               THE STATUS LINE.                  */
/*************************************************/
DECLARE MTS$MESSAGE DATA ('                    ',

DECLARE SIZE$MESSAGE DATA ('K MTS ');

/*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  */
/* THE NEXT FOUR DECLARATIONS PROVIDE THE POINTERS   */
/* UTILIZED TO CONTROL THE INPUT/OUTPUT AT EACH      */
/* TERMINAL.                                         */
/* CURSOR - SPECIFIES THE CURRENT ADDRESS WHERE THE  */
/*          CURSOR  IS TO BE DISPLAYED.              */
/* CURRENT$LINE - ADDRESS WHICH POINTS TO INITIAL BYTE */
/*          OF CURRENT USER INPUT LINE. THIS LINE HAS */
/*          NOT YET RECEIVED A 'CR' AND THUS IS NOT YET*/
/*          CONSIDERED AN INPUT BUFFER.              */
/* NEXT$CHAR - POINTS TO NEXT CHAR TO BE PROCESSED FROM*/
/*          THE INPUT BUFFER. AN INPUT IS DEFINED AS A */
/*          STRING OF ASCII CHARACTERS (FROM 1 TO 512) */
/*          WHICH HAS BEEN TERMINATED BY A 'CR' OR    */
/*          'MTS$CMD' KEY BY THE USER.               */
/* ENDSIBUFF - POINTS TO BYTE POSITION IN INPUT BUFFER */
/*          WHERE 'CR' OR 'MTS$CMD' WAS RECEIVED.    */
/*  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  *  */

DECLARE CURSOR       (8) BYTE INITIAL(0,0,0,0,0,0,0,0);
DECLARE CURRENT$LINE (8) BYTE INITIAL(0,0,0,0,0,0,0,0);
DECLARE NEXT$CHAR    (8) BYTE INITIAL(0,0,0,0,0,0,0,0);
DECLARE END$IBUFF    (8) BYTE INITIAL(0,0,0,0,0,0,0,0);

/* CAPITALIZE - SET TO ONE IF TERMINAL IN CAP MODE */

DECLARE CAPITALIZE (4) BYTE INITIAL (1,1,1,1);

/* TERM$STATUS - CONTAINS THE CURRENT STATUS OF     */
/*               EACH TERMINAL'S INPUT BUFFER,      */
/*               EITHER INPUT WAITING;              */
/*               MTS CMD READY; OR IBUFF EMPTY.     */

DECLARE TERM$STATUS (4) BYTE INITIAL ([IBUFF$EMPTY],
    [IBUFF$EMPTY],[IBUFF$EMPTY],[IBUFF$EMPTY]);

/* SWAP$POS - FOR EACH TERMINAL THERE IS A ONE BYTE */
/*            SAVE AREA WHICH IS USED DURING CURSOR */
/*            BLINKING PROCESSING                   */

DECLARE SWAP$POS (4) BYTE INITIAL ([CURSOR$CHAR],
    [CURSOR$CHAR],[CURSOR$CHAR],[CURSOR$CHAR]);

EOF
/*************************************************/
/***  TERMINAL  INTERFACE UTILITY PROCEDURES  ********/
/*************************************************/


[MACRO TRUE          '0FFH']
[MACRO FALSE         '0'    ]
```

```
[MACRO SWAPSPOS     '11FDH']

COMPARE$PTRS: PROCEDURE;
/*********************************************************/
/* COMPARES TWO POINTERS (2 BYTES EACH) TO DETERMINE*/
/* IF THEY ARE EQUAL.                                    */
/* INPUT:  DE - ADDRESS OF FIRST PTR                     */
/*         HL - ADDRESS OF SECOND PTR                    */
/* OUTPUT:  A - TRUE IF EQUAL, FALSE OTHERWISE.          */
/* CALLED BY:   KEY$COMMAND; READ$TERMINAL;              */
/*********************************************************/
    CY=0;
    IF (A=M(DE)--M(HL)) !ZERO THEN
        A=[FALSE]
    ELSE
        DO;
        DE=DE+1; HL=HL+1;
        IF (A=M(DE)--M(HL)) !ZERO THEN
            A=[FALSE]
        ELSE
            A=[TRUE];
        END;
    END COMPARE$PTRS;


CONVERT$NUMBRS$TO$ASCII: PROCEDURE;
/*********************************************************/
/* CONVERTS THE SPECIFIED NUMBER TO A DISPLAYABLE   */
/* TWO DIGIT DECIMAL NUMBER (MAX VALUE IS 99).           */
/* INPUT:  A - NUMBER TO BE CONVERTED.                   */
/* OUTPUT: B - LEFT MOST DIGIT TO BE DISPLAYED.          */
/*         C - RIGHT MOST DIGIT TO BE DISPLAYED.         */
/* CALLED BY:  SIZES$MSG; STATUS$MSG;                    */
/*********************************************************/
    B=0; C=A;
    DO WHILE (A::10) PLUS;
        B=B+1;
        C=(A=C-10);
        END;
    C=(A=A+30H);
    B=(A=B+30H);
    END CONVERT$NUMBRS$TO$ASCII;


GET$INDEX: PROCEDURE;
/*********************************************************/
/* USED TO GET THE INDEX INTO AN ADDRESS ARRAY BY   */
/* COMPUTING THE OFFSET FROM A GIVEN BASE ADDRESS.  */
/* INPUT:  A - OFFSET VALUE (NORMALLY THE TASK OR   */
/*             TERMINAL NUMBER).                         */
/*         HL - BASE ADDRESS                             */
/* OUTPUT: DE- ARRAY OFFSET                              */
/*         HL- ARRAY OFFSET (HL=DE)                      */
/*         BC- COMPUTED OFFSET                           */
/* CALLED BY:  SCROLL$DISPLAY; UPDATE$CURSOR;        */
/*             KEY$COMMAND; TERM$INPUT$CNTL;         */
/*             READ$TERMINAL; WRITE$TERMINAL         */
/*********************************************************/
    CY=0; B=0;
    C=(A=<<A);  /* SET ADDRESS OFFSET TO OFFSET*2   */
    HL=HL+BC; DE=HL;
    END GET$INDEX;


GET$VALUE: PROCEDURE;
/*********************************************************/
/* GETS A 2 BYTE VALUE FROM MEMORY.                      */
/* INPUT: DE - ADDRESS OF 2 BYTE VECTOR; THE         */
/*             CONTENTS ARE TO BE STORED IN THE      */
/*             HL REGISTER.                              */
/* OUTPUT: HL - CONTENTS OF 2 BYTE VECTOR            */
```

```
/*          DE - ADDRESS OF THE HIGH ORDER BYTE       */
/* CALLED BY:   SCROLL$DISPLAY; UPDATE$CURSOR;        */
/*              KEY$COMMAND; READ$TERMINAL;           */
/*              WRITE$TERMINAL; GET$DISPLAY$ADDR      */
/*****************************************************/
    L=(A=M(DE)); DE=DE+1; H=(A=M(DE));
    END GET$VALUE;


STORE$VALUE: PROCEDURE;
/*****************************************************/
/* STORE A 2 BYTE VALUE INTO MEMORY.                 */
/* INPUT: HL - VALUE TO BE STORED INTO MEMORY        */
/*        DE - ADDRESS OF HIGH ORDER BYTE            */
/* CALLED BY:  UPDATE$CURSOR; KEY$COMMAND;           */
/*             READ$TERMINAL; WRITE$TERMINAL;        */
/*              GET$DISPLAY$ADDR;                    */
/*****************************************************/
    M(DE)=(A=H); DE=DE-1; M(DE)=(A=L);
    END STORE$VALUE;


MOVE$BYTES: PROCEDURE;
/*****************************************************/
/* MOVES BYTES OF DATA FROM ONE MEMORY LOCATION TO   */
/* ANOTHER.                                          */
/* INPUT: BC - NUMBER OF BYTE TO BE MOVED.           */
/*        DE - STARTING MEMORY ADDRESS TO MOVE       */
/*             BYTES TO (DESTINATION).               */
/*        HL - STARTING MEMORY ADDRESS TO MOVE       */
/*             BYTES FROM (SOURCE).                  */
/* CALLED BY:  SIZE$MSG; MTS$MSG;                    */
/*****************************************************/
    REPEAT;
        M(DE)=(A=M(HL));
        DE=DE+1; HL=HL+1;
        BC=BC-1; A=0;
    UNTIL (A::B) ZERO (A::C) ZERO;
    END MOVE$BYTES;


SWAP$CURSOR: PROCEDURE;
/*****************************************************/
/* SWAP THE SPECIFIED TERMINAL'S CURRENT CURSOR      */
/* POSITION CHAR WITH THE SWAP$POS CHAR.             */
/* INPUT: A  - TERMINAL NUMBER                       */
/*        HL - DISPLAY ADDRESS OF CURSOR POSITION    */
/* CALLED BY:  BLINK$CURSORS; CHECK$CURSOR;          */
/*****************************************************/
    DE=HL;          /* SAVE DISPLAY ADDRESS          */
    B=0; C=A;
    HL=[SWAP$POS]+BC;    /* GET SWAP ADDRESS         */
    B=(A=M(DE));        /*  SWAP                     */
    M(DE)=(A=M(HL));
    M(HL)=(A=B);
    END SWAP$CURSOR;


EOF
/*****************************************************/
/******** TERMINAL INTERFACE PRIMITIVES  ************/
/*****************************************************/


[MACRO TRUE              '0FFH']
[MACRO FALSE             '0'   ]
[MACRO BLANK             '20H' ]
[MACRO IBUFF$EMPTY       '0'   ]
[MACRO DISPLAY$SIZE      '512' ]
[MACRO TERM$PORT         '03FH']
[MACRO CURSOR$CHAR       '05FH']
```

```
/*****************************************************/
/***********  INTERNAL LINKAGE MACROS  ***************/

[INT TB]              [TB := 1000H]

[MACRO STATUS$BASE   '[HEX TB + 0106H]']
[MACRO DISPLAY$BASE  '[HEX TB + 0111H]']
[MACRO CURSOR        '[HEX TB + 01D5H]']
[MACRO CURRENT$LINE  '[HEX TB + 01DDH]']
[MACRO NEXT$CHAR     '[HEX TB + 01E5H]']
[MACRO END$IBUFF     '[HEX TB + 01EDH]']
[MACRO TERM$STATUS   '[HEX TB + 01F9H]']

[MACRO GET$INDEX     '[HEX TB + 024CH]']
[MACRO GET$VALUE     '[HEX TB + 0259H]']
[MACRO STORE$VALUE   '[HEX TB + 0262H]']
[MACRO SWAP$CURSOR   '[HEX TB + 027EH]']


BLANK$DISPLAY: PROCEDURE;
/*****************************************************/
/* PLACES BLANKS INTO THE INDICATED AREA OF A        */
/* TERMINAL DISPLAY.                                 */
/* INPUT:   BC - STARTING ADDRESS (RANGE 0 TO 511)   */
/*          DE - NUMBER OF BYTES TO SET TO BLANK     */
/*          HL - BASE ADDRESS                        */
/* CALLED BY:   KEY$COMMAND; SCROLL$DISPLAY;         */
/*              CLEAR$STATUS$LINE; MTS$MSG;          */
/*****************************************************/
    HL=HL+BC;
    REPEAT;
        M(HL)=(A= [BLANK]);
        HL=HL+1;
        DE=DE-1; A=0;
    UNTIL (A::D) ZERO  (A::E) ZERO;
    END BLANK$DISPLAY;


GET$DISPLAY$ADDR: PROCEDURE;
/*****************************************************/
/* GETS A MEMORY ADDRESS IN THE DISPLAY BUFFER USING*/
/* THE DISPLAY BUFFER PTR AS OFFSET FROM THE DISPLAY*/
/* BASE ADDRESS.                                     */
/* INPUT:   BC - TERMINAL NUMBER OFFSET.             */
/*          DE - ADDRESS OF DISPLAY BUFFER PTR       */
/* OUTPUT:  HL - MEMORY ADDRESS IN DISPLAY BUFFER.   */
/*          BC - DISPLAY PTR VALUE.                  */
/* CALLED BY:   TERM$INPUT$CNTL; KEY$COMMAND;        */
/*              BLINK$CURSORS; READ$TERMINAL;        */
/*              WRITE$TERMINAL;                      */
/*****************************************************/
    DECLARE PTR(2) BYTE;
    CALL [GET$VALUE];
    PTR=HL;          /* SAVE DISPLAY PTR VALUE       */
    DE=(HL=[DISPLAY$BASE]+BC);/*GET$VALUE PARAMETER */
    BC=(HL=PTR);     /* GET DISPLAY PTR VALUE        */
    CALL [GET$VALUE]; /* GET DISPLAY BASE            */
    HL=HL+BC;        /* GET DISPLAY ADDRESS          */
    END GET$DISPLAY$ADDR;


CHECK$CURSOR: PROCEDURE;
/*****************************************************/
/* PRIOR TO CHANGING THE CURRENT CURSOR POSITION OR */
/* DISPLAYING A CHARACTER AT THE CURRENT CURSOR POS,*/
/* A CHECK IS ALWAYS MADE TO ENSURE THAT THE         */
/* CURRENT DISPLAY IS A DATA CHARACTER AND NOT THE   */
/* CURSOR ITSELF (I.E. 5FH).  IF IT IS THE CURSOR    */
/* A SWAP IS MADE.                                   */
/* INPUT: A - TERMINAL NUMBER                        */
/* CALLED BY: KEY$COMMAND; TERM$INPUT$CNTL;          */
/*            WRITE$TERMINAL;                        */
```

```
/*************************************************************/
    DECLARE T BYTE;
    T=A; HL=[CURSOR];
    CALL [GETSINDEX];
    CALL GETSDISPLAYSADDR;
    IF (A=M(HL); A::[CURSORSCHAR]) ZERO THEN
        DO;
        A=T;
        CALL [SWAPSCURSOR];
        END;
    END CHECKSCURSOR;


GETSSTATUSSADDR: PROCEDURE;
/*************************************************************/
/* GETS THE BASE ADDRESS OF THE STATUS LINE FOR THE */
/* SPECIFIED TERMINAL.                               */
/* INPUT:  A  - TERMINAL NUMBER                      */
/* OUTPUT: HL - MEMORY ADDRESS OF FIRST BYTE IN      */
/*              STATUS LINE.                         */
/* CALLED BY:   CLEARSSTATUSSLINE; MTSSMSG;          */
/*              SIZESMSG; STATUSSMSG;                */
/*************************************************************/
    HL=[STATUSSBASE];
    CALL [GETSINDEX];
    CALL [GETSVALUE];
    END GETSSTATUSSADDR;


GETSTERMSSTATUS: PROCEDURE;
/*************************************************************/
/* RETRIEVES THE TERMINAL STATUS FOR THE INDICATED  */
/* TERMINAL. TERMINAL STATUS SPECIFIES WHETHER       */
/* OR NOT THERE IS AN INPUT BUFFER OR MTS COMMAND    */
/* READY FOR PROCESSING FOR THAT TERMINAL.           */
/* INPUT:  A - TERMINAL NUMBER                       */
/* OUTPUT: A - TERMINAL STATUS                       */
/* CALLED BY:  KEYSCOMMAND; TERMINALSSTATUS;         */
/*             UPDATESCURSOR; MONITOR (MON MOD);     */
/*************************************************************/
    B=0; C=A;
    HL=[TERMSSTATUS]+BC;
    A=M(HL);
    END GETSTERMSSTATUS;


SCROLLSDISPLAY: PROCEDURE;
/*************************************************************/
/* SCROLLS THE DISPLAY FOR THE INDICATED TERMINAL.   */
/* INPUT:  A - TERMINAL NUMBER                       */
/* CALLED BY:  UPDATESCURSOR                         */
/*************************************************************/
    DECLARE TERM BYTE;
    DECLARE POS(2) BYTE;
    TERM=A; HL=[DISPLAYSBASE];
    CALL [GETSINDEX];
    CALL [GETSVALUE]; DE=HL; /*DE=DISPLAY BASE ADDR */
    POS=HL;              /* SAVE DISPLAY BASE ADDRESS*/
    BC=64;
    BC=(HL=HL+BC);       /* BC=DISPLAY BASE + 64     */
    HL=448;
    REPEAT;
        M(DE)=(A=M(BC));
        BC=BC+1; DE=DE+1;
        HL=HL-1; A=0;
    UNTIL (A::H) ZERO  (A::L) ZERO;
    BC=448;              /* SETUP PARAMETERS FOR     */
    DE=64; HL=POS;       /*  BLANKSDISPLAY PROC      */
    CALL BLANKSDISPLAY;
    END SCROLLSDISPLAY;
```

```
SENDSBEEP: PROCEDURE;
/*******************************************************/
/* SENDS A BEEP TO THE INDICATED TERMINAL.             */
/* THE FORM OF THE TERMINAL ALERT CONTROL BYTE IS:     */
/*        7  6  5  4  3  2  1  0                       */
/*       -------------------------                     */
/*       |A3|C3|A2|C2|A1|C1|A0|C0|                     */
/*       -------------------------                     */
/* WHERE:                                              */
/*       A(I) = 1; GENERATES AN ALARM AT STATION I.    */
/*       C(I) = 1; GENERATES A CLICK AT STATION I.     */
/* INPUT:  A - TERMINAL NUMBER                         */
/* CALLED BY:  KEYSCOMMAND;                            */
/*******************************************************/
    H=0; L=A;
    IF (A::0) PLUS  (A::4) MINUS THEN
        DO CASE HL;
        OUT([TERMSPORT])=(A=2);
        OUT([TERMSPORT])=(A=8);
        OUT([TERMSPORT])=(A=20H);
        OUT([TERMSPORT])=(A=80H);
        END;
    END SENDSBEEP;


SENDSCLICK: PROCEDURE;
/*************************************************************/
/* SENDS A CLICK TO THE INDICATED TERMINAL. SEE              */
/* SENDSBEEP PROC FOR DEFINITION OF TERMINAL ALERT           */
/* CONTROL BYTE.                                             */
/* INPUT:  A - TERMINAL NUMBER                               */
/* CALLED BY:  UPDATESCURSOR                                 */
/*************************************************************/
    H=0; L=A;
    IF (A::0) PLUS  (A::4) MINUS THEN
        DO CASE HL;
        OUT([TERMSPORT])=(A=1);
        OUT([TERMSPORT])=(A=4);
        OUT([TERMSPORT])=(A=10H);
        OUT([TERMSPORT])=(A=40H);
        END;
    END SENDSCLICK;


UPDATESCURSOR: PROCEDURE;
/*******************************************************/
/* CONTROLS THE UPDATING OF THE CURSOR POSITION. THE*/
/* PRIMARY CONCERN IS TO CHECK FOR SCROLLING PRIOR  */
/* TO UPDATING THE CURSOR. SCROLLING IS NOT ALLOWED */
/* IF SCROLLING WILL DESTROY ANY INPUT DATA NOT YET */
/* PROCESSED.                                       */
/* SUBPROCEDURES: CHECKSSCROLLSLOCKOUT              */
/*                UPDATESDISPLAYSPTRS               */
/* INPUT:    A  - TERMINAL NUMBER                   */
/*           HL - VALUE TO WHICH CURSOR POSITION IS */
/*                TO BE SET.                        */
/* CALLED BY:    KEYSCOMMAND; TERMSINPUTSCNTL;      */
/*               WRITESTERMINAL;                    */
/*******************************************************/
    DECLARE T BYTE;
    DECLARE POS(2) BYTE;

    CHECKSSCROLLSLOCKOUT: PROCEDURE;
    /*****************************************************/
    /* CHECKS TO SEE IF THERE IS AN INPUT BUFFER    */
    /* READY. IF SO, CHECKS TO SEE IF SCROLLING WILL*/
    /* DESTROY INPUT BUFFER. IF SO, RETURN TRUE,    */
    /* ELSE RETURN FALSE.                           */
    /* OUTPUT: A - TRUE IF SCROLLING IS LOCKED OUT  */
    /*****************************************************/
        A=T;
        CALL GETSTERMSSTATUS;
```

```
    IF (A=A-[IBUFF$EMPTY]) !ZERO THEN
        DO;      /* CHECK NEXT$CHAR PTR                    */
        A=T; HL=[NEXT$CHAR];
        CALL [GET$INDEX];/* GET NEXT$CHAR OFFSET*/
        CALL [GET$VALUE]; /* GET NEXT$CHAR VALUE*/
        BC= -64; CY=0;
        IF (HL=HL+BC) CY THEN /* SCROLL OK          */
            A=[FALSE]
        ELSE         /*  SCROLL LOCKEDOUT          */
            A=[TRUE];
        END
    ELSE         /* NO INPUT BUFFER, SCROLL OK    */
        A=[FALSE];
    END CHECK$SCROLL$LOCKOUT;


UPDATE$DISPLAY$PTRS: PROCEDURE;
/*****************************************************/
/* UPDATES ALL DISPLAY PTRS TO REFLECT THE           */
/* SCROLLING OF THE DISPLAY. USES SET$PTR TO         */
/* DECREMENT AND STORE THE POINTER VALUES.           */
/* SUBPROCEDURE: SET$PTR;                            */
/*****************************************************/

    SET$PTR: PROCEDURE;
    /*********************************************/
    /* SETS THE SPECIFIED PTR TO PTR-64, AND     */
    /* STORES THE RESULT.                        */
    /* INPUT: DE - ADDRESS OF PTR                */
    /*********************************************/
        CALL [GET$VALUE];
        BC=-64; HL=HL+BC;
        CALL [STORE$VALUE];
        END SET$PTR;

    /*********************************************/
    /* START OF UPDATE$DISPLAY$PTRS PROCESSING   */
    /*********************************************/

        /* SET CURSOR TO LEFT MARGIN OF 8TH LINE*/
        /* ON DISPLAY.                          */
    A=T; HL=[CURSOR];
    CALL [GET$INDEX];
    HL=448; DE=DE+1;
    CALL [STORE$VALUE];
        /* SET NEXT$CHAR = NEXT$CHAR - 64        */
    DE=(HL=[NEXT$CHAR]+BC);
    CALL SET$PTR;
        /* SET CURRENT$LINE = CURRENT$LINE - 64 */
    A=T; HL=[CURRENT$LINE];
    CALL [GET$INDEX];
    CALL SET$PTR;
        /* SET END$IBUFF = END$IBUFF - 64        */
    A=T; HL=[END$IBUFF];
    CALL [GET$INDEX];
    CALL SET$PTR;
    END UPDATE$DISPLAY$PTRS;

/*****************************************************/
/* START OF UPDATE$CURSOR PROCESSING                 */
/*****************************************************/

T=A;         /* SAVE INPUT TERMINAL NUMBER           */
POS=HL;      /* SAVE INPUT CURSOR POSITION           */
BC= -[DISPLAY$SIZE]; CY=0;
IF (HL=HL+BC) CY THEN   /*  SCROLLING REQUIRED  */
    DO;
    CALL CHECK$SCROLL$LOCKOUT;
    IF (A=>>A) CY THEN  /* SCROLLING LOCKED OUT */
        DO;
        A=T;
        CALL SEND$CLICK;
```

```
            END
        ELSE                    /* SCROLLING IS OK      */
            DO;
            A=T;
            CALL SCROLL$DISPLAY;
            CALL UPDATE$DISPLAY$PTRS;
            END;
        END
    ELSE    /* NO SCROLLING REQUIRED; UPDATE CURSOR */
        DO;
        A=T; HL=[CURSOR];
        CALL [GET$INDEX];   /* GET CURSOR PTR ADDR  */
        HL=POS; /* SETUP [STORE$VALUE] PARAMETERS   */
        DE=DE+1;
        CALL [STORE$VALUE]; /* UPDATE CURSOR PTR    */
        END;
    END UPDATE$CURSOR;


EOF
/*************************************************************/
/******  TERMINAL KEY PROCESSING PROCEDURES  ********/
/*************************************************************/


[MACRO TRUE                         '0FFH']
[MACRO FALSE                        '0'    ]
[MACRO BLANK                        '20H'  ]
[MACRO CURSOR$CHAR                  '5FH'  ]
[MACRO INPUT$WAITING                '0FFH']
[MACRO MTS$CMD$READY                '0F0H']
[MACRO IBUFF$EMPTY                  '0'    ]


/*************************************************************/
/**********  MTS KEY COMMAND MACROS  ****************/

[MACRO MTS$CMD                      '0A0H']
[MACRO CR                           '0DH' ]
[MACRO CHAR$DELETE                  '07FH']
[MACRO LINE$DELETE                  '015H']
[MACRO CAPITAL                      '0A1H']
[MACRO CURSOR$LEFT                  '0A2H']
[MACRO CURSOR$RIGHT                 '0A3H']
[MACRO CLEAR$SCREEN                 '0A4H']


/*************************************************************/
/*****  INTERNAL AND EXTERNAL LINKAGE MACROS  *******/

[INT GB TB]     [GB := 0]     [TB := 1000H]

[MACRO TCT$STATUS         '[HEX GB + 3E95H]']

[MACRO ASCII              '[HEX TB + 0003H]']
[MACRO DISPLAY$BASE       '[HEX TB + 0111H]']
[MACRO CURSOR             '[HEX TB + 01D5H]']
[MACRO CURRENT$LINE       '[HEX TB + 01DDH]']
[MACRO NEXT$CHAR          '[HEX TB + 01E5H]']
[MACRO END$IBUFF          '[HEX TB + 01EDH]']
[MACRO CAPITALIZE         '[HEX TB + 01F5H]']
[MACRO SWAP$POS           '[HEX TB + 01FDH]']

[MACRO COMPARE$PTRS       '[HEX TB + 0213H]']
[MACRO GET$INDEX          '[HEX TB + 024CH]']
[MACRO GET$VALUE          '[HEX TB + 0259H]']
[MACRO STORE$VALUE        '[HEX TB + 0262H]']

[MACRO BLANK$DISPLAY      '[HEX TB + 02A3H]']
[MACRO GET$DISPLAY$ADDR   '[HEX TB + 02B7H]']
[MACRO CHECK$CURSOR       '[HEX TB + 02D0H]']
[MACRO GET$TERM$STATUS    '[HEX TB + 02F9H]']
```

```
[MACRO SEND$BEEP          '[HEX TB + 033EH]']
[MACRO UPDATE$CURSOR      '[HEX TB + 03C2H]']

KEY$COMMAND: PROCEDURE;
/*************************************************************/
/* CHECKS FOR A TERMINAL KEY COMMAND FOR EVERY KEY           */
/* INTERRUPT RECEIVED.                                       */
/* KEY COMMANDS ARE:                                         */
/*      CMD          KEY          RESULT                     */
/*      ---          ---          ------                     */
/*  MTS CMD        ERROR RESET   SENDS A COMMAND TO          */
/*                               MTS FOR PROCESSING.         */
/*  CR             NEW LINE;     TERMINATES THE              */
/*                 ENTER; SHIFT  CURRENT LINE AND            */
/*                 CR; I/O CTL M;ESTABLISHES IT AS           */
/*                               THE CURRENT INPUT           */
/*                               BUFFER.                     */
/* CHAR DELETE    BACK SPACE    DELETES THE LAST            */
/*                               CHAR ENTERED.               */
/* LINE DELETE    NEXT FMAT     DELETES THE CURRENT         */
/*                               LINE.                       */
/* CAPTIALIZE     FS  C         FLIP/FLOP USED TO           */
/*                               SET OR CLEAR THE            */
/*                               TERMINAL INPUT MODE         */
/*                               TO UPPER OR LOWER           */
/*                               CASE LETTERS.               */
/* CLEAR SCREEN     FS  $       CLEARS THE 512 CHAR         */
/*                               DISPLAY BUFFER.             */
/* CURSOR LEFT      <--         MOVES CURSOR POS            */
/*                               ONE POSITION TO THE         */
/*                               LEFT.                       */
/* CUROSR RIGHT     -->         MOVES CURSOR POS            */
/*                               ONE POSITION TO THE         */
/*                               RIGHT.                      */
/* SUBPROCEDURES: ACCEPT$INPUT; CHECK$LEFT$MARGIN;           */
/*                DELETE$CHAR; CLEAR$PTRS; MTS$CMD;          */
/*                TERMINATE$CL; CARRIAGE$RETURN;             */
/*                CHAR$DELETE; LINE$DELETE; CAPITAL;         */
/*                CLEAR$SCREEN; CURSOR$LEFT;                 */
/*                CURSOR$RIGHT; CHECK$CASE;                  */
/* INPUT:   A   - TERMINAL NUMBER                            */
/*          C   - ASCII CHAR RECEIVED                        */
/* OUTPUT:  A   - TRUE IF CHAR = KEY COMMAND                 */
/* CALLED BY:     TERM$INPUT$CNTL;                           */
/*************************************************************/
    DECLARE (CHAR,T) BYTE;
    DECLARE RESPONSE BYTE;

    ACCEPT$INPUT: PROCEDURE;
    /*******************************************************/
    /* CHECKS THE TERMINAL'S CURRENT STATUS TO             */
    /* DETERMINE IF THIS NEW INPUT BUFFER SHOULD BE        */
    /* ACCEPTED.  IF SO, RETURNS TRUE, ELSE FALSE.         */
    /* OUTPUT: A  - TRUE IS INPUT CAN BE ACCEPTED.         */
    /*         HL - IF A IS TRUE THEN HL CONTAINS          */
    /*              THE ADDRESS OF TERM$STATUS.            */
    /*******************************************************/
        A=T; CALL [GET$TERM$STATUS];
        IF (A::[IBUFF$EMPTY]) !ZERO THEN
            DO; /* INPUT BUFFER HAS NOT YET BEEN          */
                /*PROCESSED;DO NOT ACCEPT NEW BUFFER*/
            A=T; CALL [SEND$BEEP];
            A=[FALSE];
            END
        ELSE
            A=[TRUE];
        END ACCEPT$INPUT;


    CHECK$LEFT$MARGIN: PROCEDURE;
    /*******************************************************/
```

```
/* CHECKS TO SEE IF CURRENT LINE IS EMPTY.       */
/* COMPARE$PTRS RETURNS THE APPROPRIATE TRUE/    */
/* FALSE VALUE IN THE A REGISTER.                */
/* INPUT: DE - ADDRESS OF CURSOR                 */
/*        BC - COMPUTED OFFSET OF TERM NBR       */
/* OUTPUT: A - RETURNED TRUE IF CURSOR IS        */
/*             PRESENTLY AT LEFT MARGIN.         */
/*************************************************/
    HL=[CURRENT$LINE]+BC;/*HL=ADD OF CURRENTLINE*/
    CALL [COMPARE$PTRS];     /* COMPARE          */
                        /* CURSOR = CURRENT$LINE */
    END CHECK$LEFT$MARGIN;


CLEAR$PTR: PROCEDURE;
/*************************************************/
/* SETS THE VALUE TO THE SPECIFIED DISPLAY       */
/* POINTER TO ZERO.                              */
/* INPUT: HL - ADDRESS OF THE DISPLAY PTR        */
/*************************************************/
    M(HL)=(A=0); HL=HL+1; M(HL)=A;
    END CLEAR$PTR;


DELETE$CHAR: PROCEDURE;
/*************************************************/
/* DECREMENTS THE CURRENT CURSOR POSITION AND    */
/* SETS NEW CURSOR POSITION DISPLAY TO BLANK.    */
/* INPUT: DE - ADDRESS OF CURSOR                 */
/*        BC - COMPUTED OFFSET OF TERMINAL NBR   */
/*************************************************/
    CALL [GET$VALUE];         /*  GET CURSOR     */
    HL=HL-1;                  /*  DECREMENT CURSOR */
    CALL [STORE$VALUE]; /* SAVE NEW CURSOR POS   */
    CALL [GET$DISPLAY$ADDR]; /* REPLACE PRESENT */
    M(HL)=(A= [BLANK]); /*       CHAR WITH BLANK */
    END DELETE$CHAR;

TERMINATE$CL: PROCEDURE;
/*************************************************/
/* TERMINATE THE CURRENT LINE.  THE SAME         */
/* PROCESSING IS DONE FOR BOTH AN MTS CMD AND    */
/* A CARRIAGE RETURN (CR) SINCE EACH SPECIFIES   */
/* THE END OF INPUT BY THE USER.                 */
/*************************************************/
            /* CHECK CHAR PRESENTLY BEING        */
            /* DISPLAYED AT CURSOR POSITION      */
            /* PRIOR TO UPDATING PTRS.           */
        A=T; CALL [CHECK$CURSOR];
            /* END OF CURRENT LINE; UPDATE       */
            /* DISPLAY POINTERS FOR NEW INPUT    */
            /* BUFFER AND NEW CURRENT LINE.      */
            /* SET END$IBUFF=CURRENT CURSOR POS */
        A=T; HL=[END$IBUFF];
        CALL [GET$INDEX];
        HL=[CURSOR]+BC;
        BC=DE; DE=HL;
        CALL [GET$VALUE];
        DE=BC+1;
        CALL [STORE$VALUE]; /* END$IBUFF=CURSOR */
            /* MOVE CURSOR TO BEGINNING OF       */
            /* NEXT  LINE. HL CONTAINS THE       */
            /* CURRENT CURSOR POSITION.          */
        BC=64; HL=HL+BC;/*ADD 64 TO CURRENT POS;*/
        L=(A=L  0C0H);/*THEN CLEAR LOWER 6 BITS*/
        A=T;
        CALL [UPDATE$CURSOR];
            /* SET CURRENT LINE = NEW CURSOR POS*/
        A=T; HL=[CURRENT$LINE];
        CALL [GET$INDEX];
        HL=[CURSOR]+BC;
        BC=DE; DE=HL;
```

```
        CALL [GETSVALUE];
        DE=BC+1;
        CALL [STORESVALUE];/*CURRENTSLINE=CURSOR*/
    END TERMINATESCL;


MTSSCMD: PROCEDURE;
/*****************************************************/
/* CHECK TO SEE IF THIS INPUT CAN BE ACCEPTED.       */
/* IF SO, SET TERMSSTATUS TO MTSSCMDSREADY AND       */
/* SET MCP BIT IN TASK CONTROL TABLE TO ENSURE       */
/* MCP IS CALLED BY THE MONITOR TO PROCESS THIS      */
/* MTS COMMAND.                                      */
/*****************************************************/
    CALL ACCEPTSINPUT;
    IF (A=>>A) CY THEN
        DO;          /* ACCEPT INPUT BUFFER          */
        M(HL)=(A=[MTSSCMDSREADY]);
        B=0; C=(A=T);
        HL=[TCTSSTATUS] + BC;
        M(HL)=(A=M(HL) \ 2);
        CALL TERMINATESCL;
        END;
    END MTSSCMD;


CARRIAGESRETURN: PROCEDURE;
/*****************************************************/
/* USER HAS TERMINATE CURRENT LINE.  CHECK TO        */
/* SEE IF NEW INPUT BUFFER CAN BE ACCEPTED.  IF      */
/* SO, SET TERMSSTATUS TO INPUTSWAITING AND          */
/* TERMINATE CURRENT LINE.                           */
/*****************************************************/
    CALL ACCEPTSINPUT;
    IF (A=>>A) CY THEN
        DO;          /*  ACCEPT INPUT BUFFER         */
        M(HL)=(A=[INPUTSWAITING]);
        CALL TERMINATESCL;
        END;
    END CARRIAGESRETURN;


CHARSDELETE: PROCEDURE;
/*****************************************************/
/* CHECK TO ENSURE THAT CURRENT LINE IS NOT          */
/* EMPTY. THEN DELETE THE PREVIOUSLY ENTERED         */
/* CHAR.                                             */
/* INPUT: DE - CURSOR OFFSET ADDRESS                 */
/*****************************************************/
    CALL CHECKSLEFTSMARGIN;
    IF (A=>>A) CY THEN
        DO;              /* CURRENT LINE EMPTY       */
        A=T;
        CALL [SENDSBEEP];
        END
    ELSE
        DO;              /*       DELETE CHAR        */
        A=T; CALL [CHECKSCURSOR];
        A=T; HL=[CURSOR];
        CALL [GETSINDEX];
        CALL DELETESCHAR;
        END;
    END CHARSDELETE;


LINESDELETE: PROCEDURE;
/*****************************************************/
/* CHECK TO ENSURE THAT CURRENT LINE IS NOT          */
/* EMPTY.  IF NOT, THEN DELETE THE CURRENT LINE.     */
/* INPUT: DE - CURSOR ADDRESS OFFSET                 */
/*****************************************************/
```

```
    CALL CHECK$LEFT$MARGIN;
    IF (A=>>A) CY THEN
        DO;                 /* CURRENT LINE EMPTY    */
        A=T;
        CALL [SEND$BEEP];
        END
    ELSE
        DO;
        A=T; CALL [CHECK$CURSOR]; A=0;
        DO WHILE (A=>>A) !CY;
        A=T; HL=[CURSOR];
        CALL [GET$INDEX]; /* GET CURSOR OFFSET  */
        CALL DELETE$CHAR;
        A=T; HL=[CURSOR];
        CALL [GET$INDEX];
        CALL CHECK$LEFT$MARGIN;
        END;
        END;
    END LINE$DELETE;


CAPITAL: PROCEDURE;
/*****************************************************/
/* SET OR CLEAR THIS TERMINALS CAPITALIZE FLAG. */
/*****************************************************/
    B=0; C=(A=T); HL=[CAPITALIZE]+BC;
    M(HL)= (A=M(HL) \\1);
    END CAPITAL;


CLEAR$SCREEN: PROCEDURE;
/*****************************************************/
/* CLEARS THE 512 BYTE DISPLAY BUFFER AND        */
/* REINITIALIZES THE DISPLAY POINTERS.           */
/* INPUT: HL - CURSOR ADDRESS                    */
/*****************************************************/
    CALL CLEAR$PTR;      /* REINITIALIZE DISPLAY */
    A=T; HL=[CURRENT$LINE];/* PTRS AND TERMINAL */
    CALL [GET$INDEX];            /* STATUS.       */
    CALL CLEAR$PTR;
    A=T; HL=[NEXT$CHAR];
    CALL [GET$INDEX];
    CALL CLEAR$PTR;
    A=T; CALL [GET$TERM$STATUS];
    M(HL)=(A=[IBUFF$EMPTY]);
                /*     CLEAR THE DISPLAY          */
    A=T; HL=[DISPLAY$BASE];
    CALL [GET$INDEX];
    CALL [GET$VALUE];  /* DISPLAY$BASE PTR IN HL*/
    BC=0; DE=512; /* SETUP INPUT PARAMETERS FOR */
                  /* [BLANK$DISPLAY] PROC       */
    CALL [BLANK$DISPLAY];
        /* RESET SWAP$POS TO CURSOR$CHAR         */
    B=0; C=(A=T);
    HL=[SWAP$POS]+BC;
    M(HL)=(A=[CURSOR$CHAR]);
    END CLEAR$SCREEN;


CURSOR$LEFT: PROCEDURE;
/*****************************************************/
/* MOVES THE CURRENT CURSOR POSITION BACK ONE.   */
/* CHECKS TO ENSURE THAT CURSOR IS NOT ALREADY   */
/* AT THE LEFT MARGIN OF CURRENT LINE.           */
/* INPUT: DE - CURSOR ADDRESS                    */
/*****************************************************/
    CALL CHECK$LEFT$MARGIN;
    IF (A=>>A) CY THEN
        DO;      /* AT LEFT MARGIN; SEND BEEP     */
        A=T;
        CALL [SEND$BEEP];
        END
```

```
    ELSE
        DO;      /* DECREMENT CURSOR                    */
        A=T; CALL [CHECK$CURSOR];
        A=T; HL=[CURSOR];
        CALL [GET$INDEX];
        CALL [GET$VALUE];
        HL=HL-1;
        CALL [STORE$VALUE];
        END;
    END CURSOR$LEFT;


CURSOR$RIGHT: PROCEDURE;
/*****************************************************/
/* MOVE THE CURRENT CURSOR POSITION FORWARD ONE.*/
/* INPUT: DE - CURSOR ADDRESS                        */
/*****************************************************/
    A=T; CALL [CHECK$CURSOR];
    A=T; HL=[CURSOR];
    CALL [GET$INDEX];
    CALL [GET$VALUE];
    HL=HL+1;              /* SETUP NEW CURSOR POS */
    A=T; CALL [UPDATE$CURSOR];
    END CURSOR$RIGHT;


CHECK$CASE: PROCEDURE;
/*****************************************************/
/* USES A CASE STATEMENT, INSTEAD OF 'ELSE DO'  */
/* TO CHECK FOR THE LAST FOUR KEY COMMANDS.  IF */
/* NOT, RESPONSE IS SET TO FALSE.  (NOTE: CASE  */
/* STMT MUST BE USED TO GET AROUND ML80'S PARSE */
/* STACK OVERFLOW, CAUSED BY TOO MANY 'IF THEN  */
/* ELSE DO' STMTS.                              */
/* INPUT: DE - CURSOR ADDRESS                   */
/*****************************************************/
    H=0; L=(A=CHAR-0A1H); /* SETUP CASE OFFSET  */
    IF (A::0) PLUS  (A::4) MINUS THEN
        DO CASE HL;
        CALL CAPITAL;
        CALL CURSOR$LEFT;
        CALL CURSOR$RIGHT;
        DO; HL=DE; CALL CLEAR$SCREEN; END;
        END
    ELSE
        RESPONSE=(A=[FALSE]);
    END CHECK$CASE;

/*****************************************************/
/* START OF KEY$COMMAND PROCESSING                   */
/*****************************************************/

T=A; CHAR=(A=C); /*  GET INPUT PARAMETERS         */
RESPONSE=(A=[TRUE]);     /* INITIALIZE RESPONSE  */
A=T; HL=[CURSOR];    /* GET$INDEX PARAMETERS      */
CALL [GET$INDEX];    /* GET CURSOR OFFSET ADDRESS*/

IF (A=CHAR-[MTS$CMD]) ZERO THEN      /* MTS CMD  */
    CALL MTS$CMD;

ELSE DO;
IF (A=CHAR-[CR]) ZERO THEN  /* CARRIAGE RETURN  */
    CALL CARRIAGE$RETURN

ELSE DO;
IF (A=CHAR-[CHAR$DELETE]) ZERO THEN
    CALL CHAR$DELETE     /*       CHAR DELETE CMD */

ELSE DO;
IF (A=CHAR-[LINE$DELETE]) ZERO THEN
    CALL LINE$DELETE     /*       DELETE LINE CMD */
```

```
    ELSE
        CALL CHECK$CASE;        /* USE CASE STMT TO      */
                                /* CHECK FOR REMAINING   */
                                /* KEY COMMANDS.         */

    END; END; END;              /* END OF ELSE DO'S      */
    A=RESPONSE;
    END KEY$COMMAND;


TERM$INPUT$CNTL: PROCEDURE;
/*************************************************************/
/* CONVERTS THE INPUT MATRIX CODE TO ASCII; CHECKS           */
/* FOR CAPIALIZATION AND CONVERTS LOWER TO UPPER             */
/* CASE LETTERS IF REQUIRED; CHECKS FOR MTS KEY              */
/* COMMANDS; IF NOT A KEY CMD THEN THE CHAR IS               */
/* DISPLAY AT THE TERMINAL AND CURSOR INCREMENTED.           */
/* INPUT: C - MATRIX CODE                                    */
/*        E - TERMINAL NUMBER                                */
/* CALLED BY: TERMINAL$HLDR (INTERRUPT MOD)                  */
/*************************************************************/
    DECLARE (CHAR,T) BYTE;

    T=(A=E);                    /* SAVE TERMINAL NUMBER      */

                        /* CONVERT MATRIX CODE TO ASCII */
    B=0; HL=[ASCII]+BC;
    CHAR=(A=M(HL));
                            /* CHECK FOR CAPITALIZATION */
    D=0; HL=[CAPITALIZE]+DE;
    IF (A=M(HL); A::0) !ZERO  (A=CHAR-61H) PLUS
        & (A=CHAR-7BH) MINUS THEN /* CONVERT TO      */
            CHAR=(A=CHAR-20H); /* UPPER CASE LETTER */

                        /* CHECK FOR ANY KEY COMMANDS   */
    C=(A=CHAR); A=T;
    CALL KEY$COMMAND;

            /* A REG RETURNED TRUE IF KEY CMD FOUND */
    IF (A=>>A) !CY THEN         /* DISPLAY CHAR      */
        DO;
        A=T; CALL [CHECK$CURSOR];
        A=T; HL=[CURSOR];
        CALL [GET$INDEX]; /* GET CURSOR OFFSET       */
        CALL [GET$DISPLAY$ADDR];
        M(HL)=(A=CHAR);
            /* UPDATE CURSOR POSITION BY ONE. BC WAS*/
            /* RETURNED FROM GET$DISPLAY$ADDR SET   */
            /* TO THE VALUE OF CURSOR.              */
        HL=BC+1;
        A=T; CALL [UPDATE$CURSOR];
        END;
    END TERM$INPUT$CNTL;


EOF
/*************************************************************/
/*****  TERMINAL INTERFACE SYSTEM FUNCTIONS  ********/
/*************************************************************/


[MACRO IBUFF$EMPTY              '0'     ]
[MACRO MTS$CMDS$READY           '0F0H'  ]
[MACRO CR                       '0DH'   ]
[MACRO LF                       '0AH'   ]



/*************************************************************/
/*****  INTERNAL AND EXTERNAL LINKAGE MACROS  *******/
```

```
[INT GB TB]     [GB := 0]  [TB := 1000H]

[MACRO TASK          'M([HEX GB + 3E90H])']

[MACRO MTS$MESSAGE   '[HEX TB + 011CH]' ]
[MACRO SIZE$MESSAGE  '[HEX TB + 01CFH]' ]
[MACRO CURSOR        '[HEX TB + 01D5H]' ]
[MACRO CURRENT$LINE  '[HEX TB + 01DDH]' ]
[MACRO NEXT$CHAR     '[HEX TB + 01E5H]' ]
[MACRO END$IBUFF     '[HEX TB + 01EDH]' ]

[MACRO COMPARE$PTRS '[HEX TB + 0213H]' ]
[MACRO CONVERT$NUMBRS$TO$ASCII '[HEX TB + 0231H]' ]
[MACRO GET$INDEX     '[HEX TB + 024CH]' ]
[MACRO GET$VALUE     '[HEX TB + 0259H]' ]
[MACRO STORE$VALUE   '[HEX TB + 0262H]' ]
[MACRO MOVE$BYTES    '[HEX TB + 026DH]' ]
[MACRO SWAP$CURSOR   '[HEX TB + 027EH]' ]

[MACRO BLANK$DISPLAY     '[HEX TB + 02A3H]']
[MACRO CHECK$CURSOR  •   '[HEX TB + 02D0H]']
[MACRO GET$DISPLAY$ADDR '[HEX TB + 02B7H]']
[MACRO GET$STATUS$ADDR  '[HEX TB + 02ECH]']
[MACRO GET$TERM$STATUS  '[HEX TB + 02F9H]']
[MACRO UPDATE$CURSOR    '[HEX TB + 03C2H]']


BLINK$CURSORS: PROCEDURE;
/*************************************************************/
/* SWAPS THE CURRENT CONTENTS OF CURSOR(I) WITH              */
/* SWAP$POS(I) FOR EACH TERMINAL (I=0 TO 3).                 */
/* CALLED BY: TIMER$HDLR (INTERRUPT MOD)                     */
/*************************************************************/
    DECLARE I BYTE;
    I=(A=3);
    REPEAT;
        HL=[CURSOR];
        CALL [GET$INDEX];
        CALL [GET$DISPLAY$ADDR];
        A=I; CALL [SWAP$CURSOR];
        I=(A=I-1);
    UNTIL (A::0) MINUS;
    END BLINK$CURSORS;


CLEAR$STATUS$LINE: PROCEDURE;
/*************************************************************/
/* CLEARS THE STATUS LINE OF THE SPECIFIED TERMINAL.         */
/* INPUT:  A - TERMINAL NUMBER                               */
/* CALLED BY:  MTS$IPL (MONITOR MOD);                        */
/*             QUIT    (SERVICE MOD);                        */
/*************************************************************/
    CALL [GET$STATUS$ADDR];
    BC=0; DE=64;          /* SETUP PARAMETERS FOR            */
    CALL [BLANK$DISPLAY]; /* BLANK$DISPLAY PROC              */
    END CLEAR$STATUS$LINE;


MTS$MSG: PROCEDURE;
/*************************************************************/
/* CONTROLS THE MTS MESSAGE DISPLAY FIELD ON THE             */
/* STATUS LINE OF THE TERMINAL SPECIFIED BY 'TASK'.          */
/* THE MTS MESSAGE FIELD STARTS AT POSITION 48 AND           */
/* UTILIZES THE REMAINING 16 BYTES FOR MTS MESSAGES          */
/* (SEE MTS$MESSAGE DATA).                                   */
/* INPUT:  E - MTS MESSAGE NUMBER.                           */
/* CALLED BY:  MTS (SERVICE MOD);                            */
/*             MINI$DISK (MONITOR MOD);                      */
/*             RECOVER   (MONITOR MOD);                      */
/*             BUMP$TASK (MONITOR MOD);                      */
/*************************************************************/
    DECLARE MSGNO BYTE;
```

```
    MSGNO=(A=E);
    A= [TASK];
    CALL [GETSSTATUSSADDR];
    BC=48;                  /* MTS MSG FIELD OFFSET FROM*/
                            /* STATUS BASE ADDRESS      */
    A=MSGNO; E=4; CY=0;
    REPEAT;                 /* COMPUTE OFFSET INTO THE  */
        A=<<A;              /* MTSSMESSAGE DATA VECTOR  */
    UNTIL (E=E-1) ZERO;
    DE=(HL=HL+BC);          /* SETUP PARAMETERS FOR     */
    B=0; C=A;               /* [MOVESBYTES] PROC        */
    HL=[MTSSMESSAGE]+BC;
    BC=16; CALL [MOVESBYTES];       /* DISPLAY MSG  */
    END MTSSMSG;


SIZESMSG: PROCEDURE;
/*************************************************************/
/* CONTROLS THE DISPLAY OF THE CURRENT MEMORY SIZE  */
/* ON THE STATUS LINE OF THE TERMINAL SPECIFIED BY  */
/* 'TASK'.  THE SIZE MESSAGE STARTS AT POSITION 40  */
/* AND HAS THE GENERAL FORMAT:                      */
/*      40      47                                  */
/*      ----------                                  */
/*       NRK MTS        E.G.  16K MTS               */
/*      ----------                                  */
/* WHERE                                            */
/*      'NR' IS THE CURRENT MEMORY SWAP SIZE        */
/*      ALLOCATED TO THAT TERMINAL USER.            */
/* THE RANGE IS FROM 0 TO 48K. THE INPUT MEMORY     */
/* SIZE NUMBER IS CONVERTED TO ASCII FOR DISPLAY.   */
/* INPUT:  A - MEMORY SIZE                          */
/* CALLED BY:  SIZE (SERVICE MOD);                  */
/*             LOGIN (SERVICE MOD);                 */
/*************************************************************/
    DECLARE MEMSSIZE BYTE;
    MEMSSIZE=A;
    A= [TASK];
    CALL [GETSSTATUSSADDR];
    BC=40;                  /* SIZE MSG FIELD OFFSET    */
    HL=HL+BC;               /* HL=STARTING ADDRESS OF   */
                            /* SIZE MSG FIELD ON STATUS */
                            /* LINE.                    */
    A=MEMSSIZE;
    CALL [CONVERTSNUMBRSTOSASCII];
    M(HL)=(A=B); HL=HL+1;   /* DISPLAY MEMORY SIZE  */
    M(HL)=(A=C); DE=HL+1;   /* SETUP PARAMETERS     */
    HL=[SIZESMESSAGE];      /* FOR [MOVESBYTES] PROC*/
    BC=6; CALL [MOVESBYTES];/* DISPLAY REST OF      */
                            /*       SIZE MESSAGE   */
    END SIZESMSG;


STATUSSMSG: PROCEDURE;
/*************************************************************/
/* CONTROLS THE STATUS DISPLAY, POSITIONS 0 THRU 39 */
/* OF THE TERMINAL STATUS LINE.  IT HAS THE         */
/* FOLLOWING GENERAL FORMAT:                        */
/*      0                                       39  */
/*      ----------------------------------------    */
/*      A=NOrB=NOrC=NOrD=NOrE=NOrF=NOrG=NOrH=NOr    */
/*      ----------------------------------------    */
/*  WHERE                                           */
/*      THE LETTER ON THE LEFT OF THE EQUAL SIGN    */
/*      SPECIFIES THE DRIVE.                        */
/*      'NO' IS THE DISK NUMBER (0-31).             */
/*      'r' IS AN OPTIONAL PARAMETER WHICH IS       */
/*      DISPLAYED WHEN THE ATTACHED DISK IS A       */
/*      RESTRICTED (r) READ ONLY DISK.              */
/*  THE TERMINAL IS SPECIFIED BY 'TASK'.            */
/* INPUT: A - ASCII CODE FOR RESTRICT (r),          */
/*            OR BLANK (SPACE).                     */
```

```
/*          B - DRIVE NUMBER (MUST BE CONVERTED TO     */
/*              A LETTER FOR DISPLAY).                 */
/*          C - DISK NUMBER (RANGE 0-31; MUST BE       */
/*              CONVERTED TO ASCII FOR DISPLAY).       */
/* CALLED BY: LOGIN      (SERVICE MOD);                */
/*            ATTACH     (SERVICE MOD);                */
/*******************************************************/
    DECLARE (RESTRICT,DRIVE$LTR,DISK$NR) BYTE;
                          /* GET INPUT PARAMETERS      */
    RESTRICT=A; DRIVE$LTR=(A=B); DISK$NR=(A=C);
    A= [TASK]; CALL [GET$STATUS$ADDR];
                /* COMPUTE THE APPROPRIATE STATUS      */
                /* BASE OFFSET TO DETERMINE WHERE      */
                /* TO DISPLAY THIS STATUS INFO         */
    C=0; B=(A=DRIVE$LTR);
    DO WHILE (A::0) !ZERO;
        C=(A=C+5);
        B=(A=B-1);
        END;
    HL=HL+BC;   /* SETUP ADDRESS FOR STATUS MSG.       */
                /* DISPLAY DRIVE LETTER                */
    M(HL)=(A=DRIVE$LTR+41H);
    HL=HL+1;
                /* DISPLAY EQUAL SIGN                  */
    M(HL)=(A='=');
    HL=HL+1; A=DISK$NR;
                /* CONVERT AND DISPLAY DISK NUMBER     */
    CALL [CONVERT$NUMBR$TO$ASCII];
    M(HL)=(A=B); HL=HL+1;
    M(HL)=(A=C); HL=HL+1;
                /* DISPLAY RESTRICT OR BLANK BYTE      */
    M(HL)=(A=RESTRICT);
    END STATUS$MSG;


TERMINAL$STATUS: PROCEDURE;
/*******************************************************/
/* PROVIDES THE INTERFACE POINT FOR OTHER MTS SYSTEM   */
/* FUNCTIONS. RETRIEVES THE CURRENT TERMINAL STATUS    */
/* FOR THE TERMINAL SPECIFIED BY 'TASK'.               */
/* OUTPUT: A - SET TO THE TERMINAL STATUS (EITHER      */
/*             INPUT$WAITING; MTS$CMD$READY; OR        */
/*             IBUFF$EMPTY) BY GET$TERM$STATUS PROC.   */
/* CALLED BY:  WRITE$TERMINAL; MTS(SERVICE MOD);       */
/*             READ$TERMINAL;                          */
/*             MTS$IPL (MONITOR MOD);                  */
/*******************************************************/
    A=[TASK];
    CALL [GET$TERM$STATUS];
    END TERMINAL$STATUS;


READ$TERMINAL: PROCEDURE;
/*******************************************************/
/* GETS THE NEXT CHAR FROM THE TERMINAL INPUT BUFFER   */
/* SPECIFIED BY 'TASK'.                                */
/* IT IS ASSUMED THAT THE CALLING PROCEDURE HAS        */
/* CHECKED TERMINAL STATUS TO ENSURE INPUT IS          */
/* WAITING PRIOR TO CALLING READ$TERMINAL.             */
/* A TEST FOR END OF IBUFF IS MADE AND IF SO, A        */
/* 'CR' CHAR IS RETURNED; THE TERMINAL STATUS IS SET   */
/* TO EMPTY; AND THE NEXT$CHAR PTR IS SET TO CURRENT   */
/* LINE.                                               */
/* IF NOT AT END OF IBUFF, THE NEXT CHAR IS RETURNED   */
/* AND THE NEXT$CHAR PTR INCREMENTED.                  */
/* OUTPUT: A - CHAR OR CR                              */
/* CALLED BY:  MTS (SERVICE MOD); MTS$IPL (MONITOR);   */
/*             MONITOR (MONITOR MOD);                  */
/*******************************************************/
    DECLARE CHAR BYTE;
    DECLARE PTR(2) BYTE;
```

```
    A= [TASK]; HL=[NEXTSCHAR];
    CALL [GETSINDEX];    /* DE=ADDR OF NEXTSCHAR PTR */
    HL=[ENDSIBUFF]+BC;/* HL=ADDR OF ENDSIBUFF PTR   */
    CALL [COMPARESPTRS]; /* NEXTSCHAR=ENDSIBUFF ??  */
    IF (A=>>A) CY THEN
        DO;              /* AT END OF IBUFF, SET          */
                         /* NEXTSCHAR = CURRENTSLINE      */
        A= [TASK]; HL=[CURRENTSLINE];
        CALL [GETSINDEX];
        BC=(HL=[NEXTSCHAR]+BC);
        CALL [GETSVALUE];    /* HL=CURRENTSLINE VALUE*/
        DE=BC+1;
        CALL [STORESVALUE];
                         /* UPDATE TERMINAL STATUS        */
        CALL TERMINALSSTATUS;    /* RETURNS HL=ADDR  */
                                 /* OF TERMSSTATUS.  */
        M(HL)=(A= [IBUFFSEMPTY]);
                         /*    RETURN 'CR' TO CALLER      */
        CHAR=(A=[CR]);
        END
    ELSE                 /* NOT AT END OF IBUFF           */
                         /* RETURN THE CHAR               */
        DO;
        A=[TASK]; HL=[NEXTSCHAR];
        CALL [GETSINDEX];    /* GET AND SAVE              */
        PTR=HL;              /* NEXTSCHAR OFFSET          */
        CALL [GETSDISPLAYSADDR];
        CHAR=(A=M(HL));      /*  RETURN CHAR              */
                         /* INCREMENT NEXTSCHAR           */
        DE=(HL=PTR+1);       /* SETUP [STORESVALUE]       */
        HL=BC+1;             /*    PARAMETERS             */
        CALL [STORESVALUE];
        END;

    A=CHAR;              /* RETURN APPROPRIATE RESPONSE  */
    END READSTERMINAL;


WRITESTERMINAL: PROCEDURE;
/*************************************************************/
/* DISPLAYS THE CHAR AT THE CURRENT CURSOR POSITION */
/* OF THE TERMINAL SPECIFIED BY 'TASK'. IT CHECKS   */
/* FOR TWO SPECIAL CHARACTERS WHICH AFFECT THE      */
/* DISPLAY CURSOR POSITION.                         */
/* 'CR' RETURNS THE CURSOR TO THE BEGINNING OF THE  */
/* CURRENT DISPLAY LINE. 'LF' MOVES THE CURSOR DOWN */
/* TO THE NEXT LINE.                                */
/* FOR ALL OTHER CHARACTERS, THE CHAR IS DISPLAYED  */
/* AND THE CURSOR POSITION INCREMENTED.             */
/* PRIOR TO OUTPUT, THE CURRENT CURSOR DISPLAY      */
/* ADDRESS IS CHECKED TO ENSURE THAT THE CURSOR CHAR*/
/* IS SAVED.  OUTPUT OF CHARACTERS IS DONE UNDER    */
/* INTERRUPT LOCKOUT TO ENSURE THAT SWAPPING BY     */
/* BLINKSCURSORS PROC IS NOT DONE.                  */
/* SUBPROCEDURE: UPDATESPTRS;                       */
/* INPUT:  E - ASCII CODE OF CHAR TO BE DISPLAYED   */
/* CALLED BY:  MTS (SERVICE MOD);                   */
/*             MTSSIPL (MONITOR MOD);               */
/*************************************************************/
    DECLARE CHAR BYTE;
    DECLARE SAVESCURSOR (2) BYTE;

    UPDATESPTRS: PROCEDURE;
    /*******************************************************/
    /* AFTER THE DISPLAY OF EACH CHAR THE CURRENT   */
    /* LINE PTR AND NEXT CHAR PTR ARE ALWAYS SET TO */
    /* NEW CURSOR POSITION.  ADDITIONALLY, THE      */
    /* TERMINAL'S STATUS IS SET TO IBUFF EMPTY.     */
    /*******************************************************/
            /*  GET CURSOR POSITION                 */
        A=[TASK]; HL=[CURSOR];
        CALL [GETSINDEX];
```

```
            /*  SET CURRENTSLINE = CURSOR            */
        BC=(HL=[CURRENTSLINE]+BC);
        CALL [GETSVALUE];    /*  HL= CURSOR VALUE    */
        DE=BC+1;
        CALL [STORESVALUE]; /* CURRENTSLINE=CURSOR  */
        SAVESCURSOR=HL;
                    /* SET NEXTSCHAR = CURSOR       */
        A=[TASK]; HL=[NEXTSCHAR];
        CALL [GETSINDEX];
        HL=SAVESCURSOR; DE=DE+1;
        CALL [STORESVALUE]; /* NEXTSCHAR=CURSOR     */
                    /* SET TERMINAL STATUS = EMPTY  */
        CALL TERMINALSSTATUS;
        M(HL)=(A=[IBUFFSEMPTY]);
        END UPDATESPTRS;

        /*********************************************/
        /*  START OF WRITESTERMINAL PROCESSING       */
        /*********************************************/

    DISABLE;
    CHAR=(A=E);
    A=[TASK]; CALL [CHECKSCURSOR];
    A=[TASK]; HL=[CURSOR];
    CALL [GETSINDEX];
    IF (A=CHAR-[CR]) ZERO THEN
        DO;                         /* CARRIAGE RETURN  */
        CALL [GETSVALUE];           /* HL=CURSOR        */
        L=(A=L  0C0H);          /* GET LEFT MARGIN      */
        END
    ELSE
        DO;
        IF (A=CHAR- [LF]) ZERO THEN
            DO;                     /* LINE FEED        */
            CALL [GETSVALUE];       /* HL=CURSOR        */
            BC=64; HL=HL+BC;
            END
        ELSE                        /* DISPLAY CHAR     */
            DO;
            SAVESCURSOR=HL;
            CALL [GETSDISPLAYSADDR];
            M(HL)=(A=CHAR);
            DE=(HL=SAVESCURSOR);
            CALL [GETSVALUE];
            HL=HL+1;            /* INCREMENT CURSOR     */
            END;
        END;
            /* HL REG HOLDS NEW CURSOR POSITION         */
    A=[TASK]; CALL [UPDATESCURSOR];
    ENABLE;
            /* UPDATE OTHER DISPLAY PTRS                */
    CALL UPDATESPTRS;
    END WRITESTERMINAL;

EOF
```

```
  /*****************************************************/
  /*  *  *  *  * MTS COMMAND PROCESSOR (MCP)   *  *  *  */
  /*****************************************************/

19FAH:

MCP:  PROCEDURE;

  /*****************************************************/
  /*  MCP IS AN INDEPENDENT MODULE OF THE MICROCOMPUTER
    TIMESHARED SYSTEM (MTS) DEVELOPED FOR THE NPS
    MICROCOMPUTER LABORATORY SYCOR 440 SYSTEM.
    THIS MODULE IS CALLED BY THE MTS MONITOR TO
    PROCESS ANY SYSTEM COMMANDS ENTERED THROUGH THE
    TERMINAL INTERFACE BY THE USER.  MTS COMMANDS ARE
    VALIDATED BY MCP AND THEN SENT TO MTS SERVICE
    CALL CONTROL MODULE FOR FURTHER PROCESSING.
    MCP IS WRITTEN IN PLM FOR TWO REASONS:
         (1) TO UTILITZE A HIGH-LEVEL LANGUAGE TO
             FACILITATE THE DESIGN AND DEBUGGING TASK
             DURING THE DEVELOPMENT OF MCP.
         (2) TO PROVIDE A PLM PROGRAM WHICH ILLUSTRATES
             THE FUNCTION CALL REQUIREMENTS
             FOR ANY USER PROGRAM TO INTERFACE WITH MTS.

    THERE ARE TWO PRIMARY DIFFERENCES BETWEEN THE MCP
    INTERFACE WITH MTS AND A USER PROGRAM/MTS INTERFACE.
         (1) ONE IS THE ENTRY PORT.  THE MTS INTERFACE PORT
             FOR USER PROGRAMS IS 2000H.  THE ENTRY PORT
             FOR MCP IS 1F00H.
         (2) THE OTHER DIFFERENCE IS THAT USER PROGRAMS
             DO NOT HAVE TO BE CONCERNED WITH SAVING
             AND RESTORING THE MTS MONITOR STACKPTR.

    MCP WILL PROCESS THE FOLLOWING SYSTEM COMMANDS
    ENTERED AT THE TERMINAL BY THE USER:

         COMMAND           PARAMETERS

         LOGIN      <DISK NUMBER> /<KEY>
         QUIT            NONE
         ATTACH     <DRIVE LETTER> <DISK NUMBER> /<KEY>
         PROTECT    <DISK NUMBER> /<KEY>
         RESTRICT   <DISK NUMBER> /<KEY>
         UNPROTECT  <DISK NUMBER> /<KEY>
         SIZE       <MEMORY SIZE>

    WHERE
         <DRIVE LETTER>- DESIGNATES A VIRTUAL FLOPPY DISK
                         DRIVE TO BE ONE OF THE LETTERS
                         A THRU H.
         <DISK NUMBER> - SPECIFIES A VIRTUAL FLOPPY DISK
                         NUMBER FROM 0-31.
         <DISK NUMBER> - SPECIFIES A DISK NUMBER FROM 0-31.
         /<KEY>        - SPECIFIES A 4 CHARACTER KEY CODE.
         <MEMORY SIZE> - SPECIFIES THE SIZE OF THE USER
                         PROGRAM AREA.
  */
  /*****************************************************/


  /*****************************************************/
  /*  *  *  *  MCP LITERAL AND DATA DECLARATIONS  *  *  */
  /*****************************************************/


  /*  *  *  *  *  MCP LITERAL DECLARATIONS  *  *  *  *  */

DECLARE
    LIT             LITERALLY 'LITERALLY',

    MTS             LIT '1F00H',/* INTERNAL MTS PORT    */
```

```
    MTS$CMD$READY   LIT '0F0H' ,
    INVALID$CMD     LIT '1'    ,

    TRUE            LIT '0FFH',
    FALSE           LIT '0'    ,
    CR              LIT '0DH'  ,
    LF              LIT '0AH'  ,
    TAB             LIT '9'    ,

    MAX$CBUFF$SIZE  LIT '64'   ,

    READ$MTS$CMD    LIT '-1'   , /* INTERNAL MTS CMD   */
    ATTACH          LIT '0'    , /*MTS SYSTEM CMD NUMBERS*/
    MTS$MSG         LIT '1'    ,
    LOGIN           LIT '2'    ,
    PROTECT         LIT '3'    ,
    QUIT            LIT '4'    ,
    RESTRICT        LIT '5'    ,
    SIZE            LIT '6'    ,
    UNPROTECT       LIT '7'    ,
    TERMINAL$STATUS LIT '8'    ,
    READ$TERMINAL   LIT '9'    ,
    WRITE$TERMINAL  LIT '10'   ;

  /*  *  *  *  *  MCP GLOBAL DECLARATIONS   *  *  *  *  */

DECLARE
    CBUFF (64)     BYTE, /* CMD BUFF FOR MTS COMMAND */
    CBUFF$LENGTH   BYTE, /* NUMBER OF CHARS IN CBUFF */
    CBUFF$PTR      BYTE; /* PTS TO NEXT CHAR IN CBUFF
                            TO BE PROCESSED          */

DECLARE
    CHAR           BYTE, /* USED FOR CHAR MANIPULATION    */
    VALUE (2)      BYTE, /* VECTOR FOR CONVERTING NUMBERS */
    PARAMETERS(6)  BYTE; /*PARAMETERS FOR MTS SYSTEM CALLS*/


  /*************************************************************/
  /*  *  *  *  *  MTS INTERFACE PROCEDURES  *  *  *  *  *  */
  /*************************************************************/


    /*********************************************************/
    /*    MTS1- PROVIDES MTS INTERFACE FOR FUNCTIONS
          WHICH DO NOT REQUIRE A RETURN VALUE.
          'F' CONTAINS THE MTS CMD NUMBER; 'A' CONTAINS
          THE PARAMETER OR ADDRESS TO LIST OF PARAMETERS.
          CALLED BY: ERROR$MSG; SEND$MTS$CMD
    */
MTS1:  PROCEDURE (F,A);
    DECLARE F BYTE, A ADDRESS;
    GO TO MTS;
    END MTS1;

    /*********************************************************/
    /*    MTS2- PROVIDES MTS INTERFACE FOR FUNCTIONS
          WHICH REQUIRE A RETURNED VALUE.
          'F' AND 'A' ARE THE SAME AS IN MTS1.
          CALLED BY: READ$CHAR; SEND$MTS$CMD
    */
MTS2:  PROCEDURE (F,A) BYTE;
    DECLARE F BYTE, A ADDRESS;
    GO TO MTS;
    END MTS2;


  /*************************************************************/
  /*  *  *  *  *  *  MCP PRIMITIVE PROCEDURES  *  *  *  */
  /*************************************************************/

READ$CHAR:  PROCEDURE BYTE;
```

```
     RETURN MTS2(READ$MTS$CMD, 0);
     END READ$CHAR;


CBUFF$NOT$EMPTY:  PROCEDURE BYTE;
     RETURN CBUFF$PTR < CBUFF$LENGTH;
     END CBUFF$NOT$EMPTY;


DEBLANK:  PROCEDURE;
     DO WHILE (CBUFF$PTR < CBUFF$LENGTH) AND
               (CBUFF(CBUFF$PTR) = ' ') OR
               (CBUFF(CBUFF$PTR) = TAB);
          CBUFF$PTR = CBUFF$PTR + 1;
          END;
     END DEBLANK;


ERROR$MSG:  PROCEDURE;
     CALL MTS1(MTS$MSG, INVALID$CMD);
     GOTO FINI;
     END ERROR$MSG;


     /*************************************************************/
     /*   FILL$CBUFF - CHECK CURRENT TERMINAL STATUS FOR
          MTS COMMAND. IF NOT, EXIT; OTHERWISE FILL THE
          COMMAND BUFFER WITH THE MTS COMMAND.
          CALLED BY: MCP MAIN CONTROL
     */
FILL$CBUFF:  PROCEDURE;
     IF MTS2(TERMINAL$STATUS,0)=MTS$CMD$READY THEN
          DO;
          CBUFF$PTR, CBUFF$LENGTH = 0;
          DO WHILE (CBUFF$LENGTH <= MAX$CBUFF$SIZE) AND
                    ((CBUFF(CBUFF$LENGTH):=READ$CHAR)<>CR);
               CBUFF$LENGTH = CBUFF$LENGTH+1;
               END;
          END;
     END FILL$CBUFF;


INITIALIZE:  PROCEDURE;
     DECLARE I BYTE;
     DO I=0 TO 5;
     PARAMETERS(I) = 0FFH;
          END;
     VALUE(0), VALUE(1) = 0;
     END INITIALIZE;


     /*************************************************************
     */
LETTER:  PROCEDURE (C) BYTE;
     DECLARE C BYTE;
     RETURN C >= 'A' AND C <= 'Z';
     END LETTER;


     /*************************************************************/
     /*   NUMBER - RETURN TRUE IF 'C' IS A NUMBER.
          CALLED BY: GET$NUMBER; GETSPARAMETERS;
                     ATTACH$CMD; SIZE$CMD
     */
NUMBER:  PROCEDURE (C) BYTE;
     DECLARE C BYTE;
     RETURN C >= '0' AND C <= '9';
     END NUMBER;


     /*************************************************************/
     /*   READ$CMD$LINE - READS CHAR FROM COMMAND BUFFER;
```

```
          CONVERTS LETTERS FROM LOWER TO UPPER CASE,
          IF REQUIRED.
          CALLED BY: GETSKEY; GETSNUMBER; GETSPARAMETERS;
                     ATTACHSCMD; SIZESCMD; MCP MAIN CONTROL
READSCMDSLINE:  PROCEDURE BYTE;
     DECLARE C BYTE;
     IF (C:=CBUFF(CBUFFSPTR)) >= 61H /* LOWER CASE A      */
        AND C <= 7AH THEN    .       /*  LOWER CASE Z     */
          C = C AND 5FH;     /* CONVERT TO UPPER CASE     */
     CBUFFSPTR = CBUFFSPTR+1;
     RETURN C;
     END READSCMDSLINE;


     /*************************************************************/
     /*    SCANSTOSBLANK - SCAN TO NEXT BLANK OR TAB CHAR.
           CALLED BY: MCP MAIN CONTROL
     */
SCANSTOSBLANK:  PROCEDURE;
     DO WHILE (CBUFFSPTR<CBUFFSLENGTH) AND
              (CBUFF(CBUFFSPTR)<>' ') AND
              (CBUFF(CBUFFSPTR)<>TAB);
          CBUFFSPTR = CBUFFSPTR+1;
          END;
     END SCANSTOSBLANK;


     /*************************************************************/
     /*    SENDSMTSSCMD - 'CMD' CONTAINS THE MTS CMD NUMBER
           AND 'PARAMETER' CONTAINS THE ACTUAL PARAMETER
           OR THE ADDRESS OF THE ACTUAL PARAMETER LIST.
           TWO MTS SYSTEM CALLS ARE MADE:
           (1) MTS2 - PROCESS SYSTEM CMD, WHICH RETURNS
                      A RESPONSE.
           (2) MTS1 - DISPLAY RESPONSE AT USER TERMINAL.
     */
SENDSMTSSCMD:  PROCEDURE (CMD,PARAMETER);
     DECLARE CMD BYTE, PARAMETER ADDRESS;
     CALL MTS1(MTSSMSG, MTS2(CMD,PARAMETER));
     END SENDSMTSSCMD;


  /*************************************************************/
  /*  *  *  *  *  *  UTILITY PROCEDURES  *  *  *  *  *  *  *  */
  /*************************************************************/


     /*************************************************************/
     /*    CONVERTSVALUE - CONVERT ASCII CODE IN 'VALUE'
           TO APPROPRIATE BINARY VALUE.  THE RANGE OF VALUES
           TO BE CONVERTED ARE FROM 0-31.
           CALLED BY: GETSNUMBER;
     */
CONVERTSVALUE:  PROCEDURE BYTE;
     IF VALUE(1) = 0 THEN /*ONLY A SINCLE DIGIT TO CONVERT*/
          RETURN VALUE(0)-30H;
     ELSE                      /* TWO DIGITS TO CONVERT       */
          RETURN ((VALUE(0)-30H)*10+(VALUE(1)-30H));
     END CONVERTSVALUE;


     /*************************************************************/
     /*    GETSKEY - GET THE <KEY> PARAMETER, IF ENTERED;
           STORE KEY IN THE PARAMETER LIST STARTING AT I.
           CALLED BY: ATTACHSCMD; GETSPARAMETERS;
     */
GETSKEY:  PROCEDURE (I);
     DECLARE I BYTE;
     IF CBUFFSNOTSEMPTY THEN   /*NEXT CHAR MUST BE '/'     */
          DO;
          IF (CHAR:=READSCMDSLINE) = '/' THEN
               DO WHILE (I<6) AND CBUFFSNOTSEMPTY;
```

```
                    PARAMETERS(I)=READ$CMD$LINE;
                    I=I+1;
               END;
          ELSE                    /* '/' IS USED TO INDICATE */
               CALL ERROR$MSG; /* <KEY> AND IS REQUIRED.  */
          END;
     END GET$KEY;


     /*************************************************************/
     /*    GET$NUMBER - GET <DISK NUMBER> PARAMETER AND
          CONVERT IT FROM ASCII TO BINARY. STORE THE
          RESULT IN PARAMETER LIST AT 'I'. UPON ENTRY,
          'CHAR' HOLDS THE FIRST DIGIT.
          CALLED BY: ATTACH$CMD; GET$PARAMETERS;
     */
GET$NUMBER:  PROCEDURE (I);
     DECLARE I BYTE;
     VALUE(0) = CHAR;
     IF CBUFF$NOT$EMPTY AND
        NUMBER(CHAR:=READ$CMD$LINE)
        THEN                  /*  TWO DIGIT DISK NUMBER       */
          VALUE(1) = CHAR;
     PARAMETERS(I) = CONVERT$VALUE;
     END GET$NUMBER;


     /*************************************************************/
     /*    GET$PARAMETERS - USED TO GET THE <DISK NUMBER>
          AND <KEY> PARAMETERS. GENERATES ERROR MSG IF
          NEXT CHAR IS NOT A NUMBER.
          CALLED BY: LOGIN$CMD; GET$REQUIRED$PARAMETERS;
     */
GET$PARAMETERS:  PROCEDURE;
     IF NUMBER(CHAR:=READ$CMD$LINE) THEN
          DO;
          CALL GET$NUMBER(0);
          CALL DEBLANK;
          CALL GET$KEY(1);
          END;
     ELSE
          CALL ERROR$MSG;
     END GET$PARAMETERS;


     /*************************************************************/
     /*    GET$REQUIRED$PARAMETERS - GETS THE <DISK NUMBER>
          AND <KEY> PARAMETERS FOR THOSE SYSTEM CMDS FOR
          WHICH THESE PARAMETERS ARE REQUIRED (NOT
          OPTIONAL). GENERATES ERROR MSG IF PARAMETERS
          ARE NOT THERE.
          CALLED BY:PROTECT$CMD; RESTRICT$CMD; UNPROTECT$CMD
     */
GET$REQUIRED$PARAMETERS:  PROCEDURE;
     IF CBUFF$NOT$EMPTY THEN
          CALL GET$PARAMETERS;
     ELSE
          CALL ERROR$MSG;
     END GET$REQUIRED$PARAMETERS;


   /*************************************************************/
   /*  *  *  *  *  *  SYSTEM CMD PROCEDURES  *  *  *  *  *  */
   /*************************************************************/


     /*************************************************************/
     /*    ATTACH$CMD -
          FORM:  ATTACH <DRIVE LTR> <DISK NR> /<KEY>
          ALL THE PARAMETERS ARE OPTIONAL, HOWEVER
          <KEY> CAN NOT APPEAR WITHOUT IT'S ASSOCIATED
          <DISK NUMBER>. WHEN PARAMETERS ARE ENTERED
```

```
          THEY MUST BE IN THE ORDER INDICATED.
          CALLED BY: MCP MAIN CONTROL
     */
ATTACHSCMD:  PROCEDURE;
     IF CBUFFSNOTSEMPTY THEN
          DO;
          CHAR = READSCMDSLINE;
          IF LETTER(CHAR) THEN       /* <DRIVE LETTER>      */
               DO;
               PARAMETERS(0)=CHAR-41H;/*CONVERT TO BINARY*/
               CALL DEBLANK;
               IF CBUFFSNOTSEMPTY THEN /* MORE PARAMETERS */
                    CHAR = READSCMDSLINE;
               END;
          IF NUMBER(CHAR) THEN       /* <DISK NUMBER>       */
               DO;
               CALL GETSNUMBER(1);
               CALL DEBLANK;
               CALL GETSKEY(2);      /*  <KEY>              */
               CALL DEBLANK;
               END;
          END;
     IF CBUFFSNOTSEMPTY THEN
          CALL ERRORSMSG;
     ELSE
          CALL SENDSMTSSCMD(ATTACH,.PARAMETERS);
     END ATTACHSCMD;


     /*************************************************************/
     /*   LOGINSCMD -
          FORM: LOGIN <DISK NUMBER> /<KEY>
          THE PARAMETERS ARE OPTIONAL BUT <KEY> CAN NOT
          APPEAR WITHOUT <DISK NUMBER>.
          CALLED BY: MCP MAIN CONTROL
     */
LOGINSCMD:  PROCEDURE;
     IF CBUFFSNOTSEMPTY THEN
          CALL GETSPARAMETERS;
     CALL SENDSMTSSCMD(LOGIN,.PARAMETERS);
     END LOGINSCMD;


     /*************************************************************/
     /*   PROTECTSCMD -
          FORM: PROTECT <DISK NUMBER> /<KEY>
          THE PARAMETERS ARE REQUIRED.
          CALLED BY: MCP MAIN CONTROL
     */
PROTECTSCMD:  PROCEDURE;
     CALL GETSREQUIREDSPARAMETERS;
     CALL SENDSMTSSCMD(PROTECT, .PARAMETERS);
     END PROTECTSCMD;


     /*************************************************************/
     /*   QUITSCMD - FORM:  QUIT
          NO PARAMETERS.
          CALLED BY: MCP MAIN CONTROL
     */
QUITSCMD:  PROCEDURE;
     CALL SENDSMTSSCMD(QUIT,0);
     END QUITSCMD;


     /*************************************************************/
     /*   RESTRICTSCMD -
          FORM: RESTRICT <DISK NUMBER> /<KEY>
          THE PARAMETERS ARE REQUIRED.
          CALLED BY: MCP MAIN CONTROL
     */
RESTRICTSCMD:  PROCEDURE;
```

```
    CALL GETSREQUIREDSPARAMETERS;
    CALL SENDSMTSSCMD(RESTRICT,.PARAMETERS);
    END RESTRICTSCMD;


    /*****************************************************/
    /*    UNPROTECTSCMD -
          FORM: UNPROTECT <DISK NUMBER> /<KEY>
          THE PARAMETERS ARE REQUIRED.
          CALLED BY: MCP MAIN CONTROL
    */
UNPROTECTSCMD:  PROCEDURE;
    CALL GETSREQUIREDSPARAMETERS;
    CALL SENDSMTSSCMD(UNPROTECT,.PARAMETERS);
    END UNPROTECTSCMD;


    /*****************************************************/
    /*    SIZESCMD -
          FORM: SIZE <MEMORY SIZE>
          THE PARAMETER IS REQUIRED.
          CALLED BY: MCP MAIN CONTROL
    */
SIZESCMD:  PROCEDURE;
    IF CBUFFSNOTSEMPTY THEN
          DO;
          IF NUMBER(CHAR:=READSCMDSLINE) THEN
               DO;          /* GET MEMORY SIZE PARAMETER  */
               CALL GETSNUMBER(0);
               CALL SENDSMTSSCMD(SIZE, PARAMETERS(0));
               END;
          ELSE            /* PARAMETER MUST BE NUMBER      */
               CALL ERRORSMSG; /* SPECIFYING MEMORY SIZE   */
          END;
    ELSE                  /* CBUFF EMPTY - ERROR           */
          CALL ERRORSMSG; /*           PARAMETER REQUIRED  */
    END SIZESCMD;


  /*************************************************************/
  /*  *  *  MTS COMMAND PROCESSOR (MCP) MAIN CONTROL *  */
  /*  *  *  CALLED BY:  MTS MONITOR MODULE           *  */
  /*************************************************************/


DECLARE STACK (20) ADDRESS,OLDSP ADDRESS;

OLDSP = STACKPTR;          /* SAVE MTS STACK POINTER       */
STACKPTR = .STACK(LENGTH(STACK));  /* SETUP MCP STACKPTR */

CALL INITIALIZE;           /* INITIALIZE DATA STRUCTURES   */
CALL FILLSCBUFF;           /* GET MTS COMMAND              */
CALL DEBLANK;              /* SCAN TO FIRST NONBLANK CHAR  */
IF CBUFFSNOTSEMPTY THEN    /* PROCESS CMD BUFFER           */
    DO;
    CHAR = READSCMDSLINE;/* GET FIRST LETTER OF CMD       */
    CALL SCANSTOSBLANK;  /* SCAN TO NEXT BLANK, BECAUSE
                            ONLY THE FIRST LETTER IS USED
                            TO DETERMINE THE CMD          */
    CALL DEBLANK;

    IF CHAR = 'A' THEN                          /* ATTACH   */
         CALL ATTACHSCMD;
    ELSE
         DO;
         IF CHAR = 'L' THEN                     /* LOGIN    */
              CALL LOGINSCMD;
         ELSE
              DO;
              IF CHAR = 'P' THEN                /* PROTECT  */
                   CALL PROTECTSCMD;
              ELSE
```

```
                        DO;
                        IF CHAR = 'Q' THEN          /* QUIT      */
                              CALL QUIT$CMD;
                        ELSE
                              DO;
                              IF CHAR = 'R' THEN    /* RESTRICT */
                                    CALL RESTRICT$CMD;
                              ELSE
                                    DO;
                                    IF CHAR='S' THEN /* SIZE     */
                                         CALL SIZE$CMD;
                                    ELSE
                                      DO;
                                      IF CHAR='U' THEN/*UNPROTECT*/
                                           CALL UNPROTECT$CMD;
                                      ELSE
                                           CALL ERROR$MSG;
                                      END;
                                    END;
                              END;
                        END;
                  END;
            END;
      END;

FINI:
STACKPTR = OLDSP;   /* RESTORE MTS MONITOR STACKPTR        */

END MCP;


EOF
```

LIST OF REFERENCES

1. Bullock, D. R. and Brown, K. J., Hardware Characteristics of the Sycor 440 Clustered Terminal Processing System, unpublished research notes, Naval Postgraduate School, Monterey, CA, March 1977.

2. Digital Research, An Introduction to CP/M Features and Facilities, Pacific Grove, CA 93950, 1976.

3. Digital Research, CP/M Dynamic Debugging Tool (DDT) User's Guide, Pacific Grove, CA 93950, 1976.

4. Digital Research, CP/M Interface Guide, Pacific Grove, CA 93950, 1976.

5. Digital Research, CP/M System Alteration Guide, Pacific Grove, CA 93950, 1976.

6. Digital Research, ED: Context Editor for the CP/M Disk System User's Manual, Pacific Grove, CA 93950, 1975.

7. Hilburn, John L. and Julich, Paul N., Microcomputers / Microprocessors: Hardware, Software, and Applications, Prentice-Hall, 1976.

8. Intel Corp., 8080 Assembly Language Programming Manual, Santa Clara, CA 95051, 1976.

9. Intel Corp., 8008 and 8080 PL/M Programming Manual, Santa Clara, CA 95051, 1976.

10. Madnick, Stuart E. and Donovan, John J., Operating Systems, McGraw-Hill, 1974.

11. Meyer, R. A. and Seawright, L. H., A Virtual Machine Time-Sharing System, IBM System Journal, No. 3, p. 199-217, 1970.

12. Pedroso, L. R. B., ML80: A Structured Machine-Oriented Microcomputer Programming Language, M.S. Thesis, Naval Postgraduate School, Monterey, CA, December 1975.

13. Sycor Inc., Model 340/340D Intelligent Communications Terminal Operator's Manual, Nr TD34003-275, Ann Arbor, Michigan, October 1974.

14. Sycor Inc., User's Guide to the 440 System, Nr UG4400-2, Ann Arbor, Michigan, November 1976.

15. Sycor Inc., 8080 Debug User's Guide, Spec. 950706, Rev B, Ann Arbor, Michigan, October 1975.

16. Watson, Richard W., Timesharing System Design Concepts, McGraw-Hill, 1970.

Thesis
B816
c.1
Brown
169868
A shared environment for microcomputer system development.